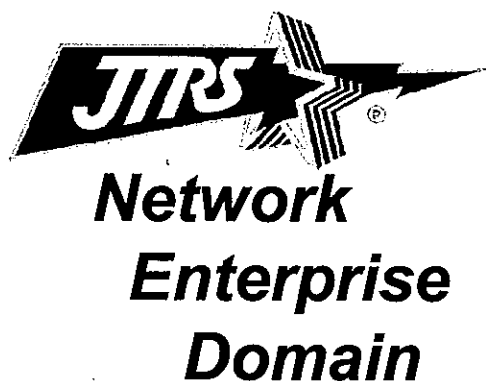


Joint Tactical Radio System Network Enterprise Domain Test & Evaluation

Waveform Portability Guidelines

VERSION 1.2.1

28 December 2009



Prepared for:

Joint Tactical Radio System Network Enterprise Domain
33000 Nixie Way, San Diego, CA 92147

Prepared by:

Network Enterprise Domain Test & Evaluation
Space and Naval Warfare Systems Center Atlantic
P.O. Box 190022, North Charleston, SC 29419-9022


Distribution A: Approved for public release; distribution is unlimited.

JAN 11 2010

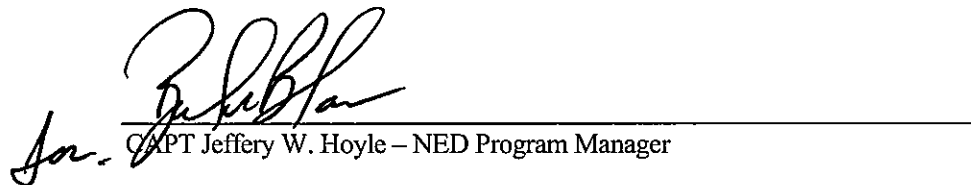
Signatures of Concurrence



Mr. Richard Anderson -- NED T&E Program Manager



Mr. Dean Nathans -- NED Chief Engineer



for. CAPT Jeffery W. Hoyle -- NED Program Manager

Change History

Status / Version	Date Modified	Author	Reason for Change
V 1.0	30 April 2007	Lane Anderson	First release
V 1.1	26 June 2008	Lane Anderson	Includes feedback from industry, input deferred from V1.0, and more lessons learned
V1.2	28 December 2009	Lane Anderson	Periodic updates
V1.2.1	28 December 2009	Lane Anderson	Changes necessary for public release

Table of Contents

1	Introduction	1
1.1	Notes.....	1
1.2	Background	1
1.2.1	Guidelines Development.....	2
1.2.2	Portability versus Optimality.....	2
1.2.3	Portability Assessment Role in the Acquisition Process.....	2
1.2.4	Portability Assessment Objective	3
1.2.5	Portability Measurement Overview	3
1.3	Organizational Chain.....	4
1.4	Target Audiences and Document Usage.....	4
2	Referenced or Related Documents	7
3	Development Guidelines	9
3.1	General Development Guidelines	9
3.1.1	Consideration of Intended Use of the Waveform.....	11
3.1.2	Use of Third-Party Software	11
3.1.3	Development Tools and Debug Code	11
3.1.4	Modeling	12
3.1.4.1	PIM/PSM Modeling and Transformation	12
3.1.4.2	Signal Processing Code Modeling.....	12
3.1.4.3	GPP Code Modeling	13
3.1.5	Waveform Architecture	13
3.1.6	Inline Documentation	15
3.2	C/C++ Programming Practices.....	16
3.3	GPP Guidelines	19
3.3.1	GPP Performance.....	19
3.3.2	POSIX Compliance.....	19
3.3.3	Use of JPEO Standard APIs	19
3.3.4	Device Interface Abstractions.....	19
3.4	DSP Guidelines	20
3.5	FPGA Guidelines	21
3.6	Domain Profile Guidelines	23
3.7	Documentation Guidelines	23
3.7.1	Waveform Design Specification	24
3.7.2	Software Requirements Specification	24
3.7.3	Software Design Description	25
3.7.4	Interface Design Descriptions.....	27
3.7.5	Software Version Description	27
3.7.6	Waveform Porting Plan	28
3.7.7	Waveform Porting Report.....	28
4	Portability Assessment Background	30

4.1	Portability Qualities	30
4.2	Assessment Methodology	31
4.2.1	Laboratory Analysis.....	31
4.2.2	Waveform Porting	32
4.2.3	Static Assessment	32
4.3	Assessment Results	32
Appendix A	Acronym List	A-1
Appendix B	Specific Terminology	B-1
Appendix C	Acquisition Guidelines.....	C-1
Appendix D	Modeling.....	D-1

Figures and Tables

Figure 1-1. Waveform Timeline, Part 1.....	5
Figure 1-2. Waveform Timeline, Part 2.....	6
Figure 3-1. Example of Platform Abstraction	20
Table 3-1. Portable C/C++ Language Features.....	16
Table 3-2. Non-Portable C/C++ Language Features	17
Table 4-1. Portability Quality Rationale	30

1 Introduction

The Joint Tactical Radio System (JTRS) Network Enterprise Domain Test and Evaluation (NED T&E) organization will conduct waveform portability assessments on all JTRS waveforms and other waveforms intended for hosting on JTRS radio platforms. All waveform portability assessments will follow the procedures defined in the NED T&E Waveform Portability Assessment Procedures (WPAP) [1], which are derived from the guidelines in this document. This document describes the practices that both the Government and waveform developer should follow to ensure that the delivered waveform exhibits the maximum portability achievable within the required performance parameters of that waveform.

1.1 Notes

- Throughout this document, the term “the Government” indicates United States Government personnel and its designated representatives.
- Several places in this document contain the phrase “as practical” or “if practical.” In such cases, the Government will determine what is practical.
- Throughout this document, the term “software” includes C/C++, other high-level languages, such as JAVA, Interface Definition Language (IDL), assembly language, and Hardware Definition Language (HDL). The reader should be aware that HDL “software” requires such things as static timing analysis, interface diagrams, etc., that other software does not.
- The portability assessments assume that each waveform has at least one target platform identified for it by the Joint Program Executive Office (JPEO) or the NED. If no such target has been identified, the Portability Assessment Team (PAT) will select one for its assessments.

1.2 Background

The JTRS Charter [2] lists portability as a JTRS Program objective, and the JTRS Operational Requirements Document (ORD) [3] defines the term waveform as follows: *“A waveform is the representation of a signal as a plot of amplitude versus time. In general usage, the term waveform refers to a known set of characteristics, e.g. SINCGARS or EPLRS “waveforms.” In JTR System usage, the term waveform is used to describe the entire set of radio functions that occur from the user input to the RF [Radio Frequency] output and vice versa. A JTR System “waveform” is implemented as a re-useable, portable, executable software application that is independent of the JTR System operating system, middleware, and hardware.”*

JTRS is unique among Department of Defense (DoD) acquisition programs in establishing the programmatic goal of procuring waveform application software in a form that can be ported (rehosted or transferred) to different Joint Tactical Radio (JTR) platforms at a cost considerably lower than that for new development. The portability of the waveform software across multiple JTR platforms is necessary for the JTRS Program to achieve the following goals:

- Reduced cost through maximized reuse of waveform software across multiple platforms
- Faster insertion of new technologies
- Interoperability of radio systems between services
- Reduced training requirements due to commonality of platforms

All new JTRS waveform acquisition contracts shall require developers to follow the guidelines set forth in this document to the maximum extent practical while maintaining the performance requirements of the

waveform. The Government does not intend this to be a burden to the developer, but rather an aid to successful ports of the waveform to the designated target platforms. This document details guidelines that should be a part of the natural development process for software and documentation necessary to enable portability success.

Unlike previous DoD software acquisitions, the product is not solely an executable binary; instead, it is a set of source files proven to operate according to a set of performance and functional requirements in a specified environment or set of environments. Further, as the Government or its representatives has the responsibility for fielding the waveform on one or more JTRS radios, the waveform design and implementation documentation, including models, is considered as important to the success of the program as the software itself. The documentation and models assume this degree of importance because one of the ultimate goals of JTRS portability is to enable a third-party organization to port or extend/enhance a waveform without support from the original developer. Beyond just capturing the design of the waveform, these documentation and modeling artifacts must also capture the motivations. This will allow the Waveform Integrator (WI) to understand the rationale behind certain engineering decisions, and how those decisions might need reconsideration for optimizing the waveform for operation on differing platforms.

Together, the source code, design documents, and models enable the Government's assessment process, which determines the gap between the requirement for waveform portability and the waveform developer's design and implementation.

1.2.1 Guidelines Development

The information in this document is a composite based upon the practical experiences of the NED T&E organization, other contributors to this document – both from inside and outside of the JTRS Program, and highly regarded industry references.

1.2.2 Portability versus Optimality

The two concepts of portability and optimality are both very important to the success of Software Defined Radio (SDR) programs such as JTRS, but they can often be in opposition to one another. Portability calls for the most generic application possible, so that it can move from one environment to another with minimal changes. Conversely, optimality calls for an application tailored to a specific platform in order to maximize efficiency and performance. Satisfying all of the requirements for the waveform requires that the developer reach a compromise between these two concepts. The JTRS program will target each waveform at a subset of the specified platforms, and the waveforms should be developed specifically for those characteristics that are common to all of their target platforms. The waveform design must be generic enough so that those characteristics that differ among target platforms do not present an unacceptable hindrance to porting between them. During the ports, the combined waveform and platform can be optimized to the degree necessary for best performance.

1.2.3 Portability Assessment Role in the Acquisition Process

The PAT exists to ensure that waveforms acquired by the Government fulfill the necessary portability objectives for the overall success of the JTRS Program. For the PAT to provide the maximum assurance of this success, it is important that the team be involved at the beginning of the acquisition process. In doing so, the PAT can:

- Provide the Acquisition Lead with guidance regarding the content of the Request For Proposal (RFP) and Statement of Work (SOW).
- Act as a resource to the waveform developer by providing:
 - A clear understanding of the portability constraints for the waveform under development

- Help in understanding the characteristics of the platforms on which the waveform will be hosted
- Work with the waveform developer during the design and development phases to ensure the early identification and avoidance of potential barriers to portability.

To perform these services, the PAT must consist of a collection of skilled engineers whose cumulative knowledge and experience covers the tools and techniques required for waveform development and porting, as well as a thorough knowledge of the intended target platforms. Refer to Appendix C for more details on the composition of the PAT.

Some waveform acquisitions have already progressed beyond the point at which the PAT can provide maximum benefit. For these programs, this document can still add value by serving as guidance for those phases of the acquisition that have not yet occurred. The Government will address any contractual issues associated with the application of these guidelines separately for each waveform.

Waveform developers must understand that the burden of proof is on them. The Government has contracted them to develop a portable waveform, and they must present a compelling case that they have done so with the implementation and documentation.

1.2.4 Portability Assessment Objective

The objective of the portability assessment process is to evaluate waveform portability by measuring whether the waveform under investigation can be made to load, instantiate, and execute correctly in the specified target JTRS set environment(s) with minimal changes. The waveform portability assessment is based upon qualitative and quantitative measures of the waveform design and implementation in a specific software and hardware environment context for determining:

- The degree to which the integrated product (ported waveform, software infrastructure, and hardware) fulfills the stated requirements
- The ease and degree by which the waveform software, HDL, interfaces, etc., developed for correct execution in one JTRS environment will execute properly within another, different JTRS environment after porting for less than the total cost to redevelop the waveform

The assessment criteria, contexts, and processes enable the above measurements and allow the PAT to assess the waveform's ability to port to all target platforms. This process requires that the PAT assess the qualities of the waveform detailed in Section 4, Table 4-1.

The paragraphs above state several times that the issue addressed here is portability *in the context of the JTRS Program*. The intent of these guidelines is to aid in the development and acquisition of JTRS waveforms that port successfully to platforms identified by the U.S. Government and its allies, although these guidelines will benefit any SDR program where portability is desired. Due to the focus on SDR in general, and JTRS in particular, it is possible that some of the guidelines contained herein will run counter to general software portability concepts. That said, the PAT will note any potential barriers to the broader definition of portability (i.e., outside of the restrictions of the JTRS program) that it identifies during an assessment. Such notes may provide value if the waveform is later ported to a platform that differs from those identified during the original waveform development program. The PAT will guide the development of JTRS waveforms toward general portability where it does not adversely affect JTRS program costs, functionality, or performance.

1.2.5 Portability Measurement Overview

The PAT measures waveform portability in accordance with the WPAP [1]. This assessment is a combination of laboratory analysis of the software components and static analysis of the waveform source

code and documentation. It also includes porting some or all waveform components to a target platform. Section 4.2 includes a more detailed overview of these activities.

1.3 Organizational Chain

The lead office for the JTRS Program is the JPEO JTRS. The NED Program Management Office (PMO) acquires JTRS waveforms and network enterprise services for the JPEO JTRS Enterprise. NED T&E is NED's test and evaluation arm responsible for assessing waveform portability and performance. Within NED, NED T&E reports to the NED Chief Engineer.

1.4 Target Audiences and Document Usage

This document targets several distinct audiences:

- The Government program office in charge of the waveform acquisition should use this document for:
 - Guidance regarding how portability-related aspects of the acquisition process are managed from start to finish, increasing the likelihood that the final delivered waveform is portable
 - Adherence to the contract guidelines to ensure that the Government and the waveform developer both produce the quality artifacts necessary to achieve the stated goal of portability
- The waveform developer should use this document for detailed information on:
 - Design concepts and practices that promote development of a portable waveform application
 - Programming practices that will enhance the portability of the developed software, thus enabling the port to a target platform by a future WI
 - Documentation and models required to provide a future WI with sufficient information to understand the waveform structure and functionality well enough to port it
 - Documentation and models that provide the WI and the PAT with the rationale behind engineering decisions
 - Documentation and models required to provide the PAT with sufficient information to perform the portability assessment
 - An understanding of the process used by the Government to assess portability.
- The PAT uses this document as the source reference on waveform portability criteria used in the waveform assessment process. The procedures described in the WPAP [1] derive from the guidelines in this document.
- The WI uses this document to provide a guide to the available information and assessment criteria on the waveform that they have been contracted to port.

Figure 1-1 and Figure 1-2 illustrate a notional timeline for the waveform acquisition activities related to portability and its assessment, highlighting in red (white text on dark background for black and white printout) those steps in which the formal procedures described in this document take place. Those activities on the NED T&E timeline that are not in red still involve NED T&E participation and evaluation of waveform design for portability. In those cases, NED T&E participates to provide guidance to the developer through participation in design reviews, review of portability related deliverable documentation, and interim code drop assessments. A final assessment is made of the Formal Qualification Testing (FQT) drop and is placed in the JPEO Information Repository (IR) as guidance for the WI. Note that the Development Phase will usually result in multiple drops of waveform software to the Government. This

will allow for multiple reviews of code at several stages in the development process, allowing any issues to be addressed much earlier than if only the final completed waveform is assessed. Some variances in this timeline are to be expected from one waveform acquisition to another.

Note: These figures do not include subsequent ports of the waveform to platforms for use in the field.

The following acronyms are marked with an asterisk (*) in the figures as there is not adequate space to expand them there: SRR – Software Requirements Review, PDR – Preliminary Design Review, CDR – Critical Design Review, TRR – Test Readiness Review, FQT – Formal Qualification Testing, PRR – Porting Readiness Review, and PPVT – Post-Port Verification Test.

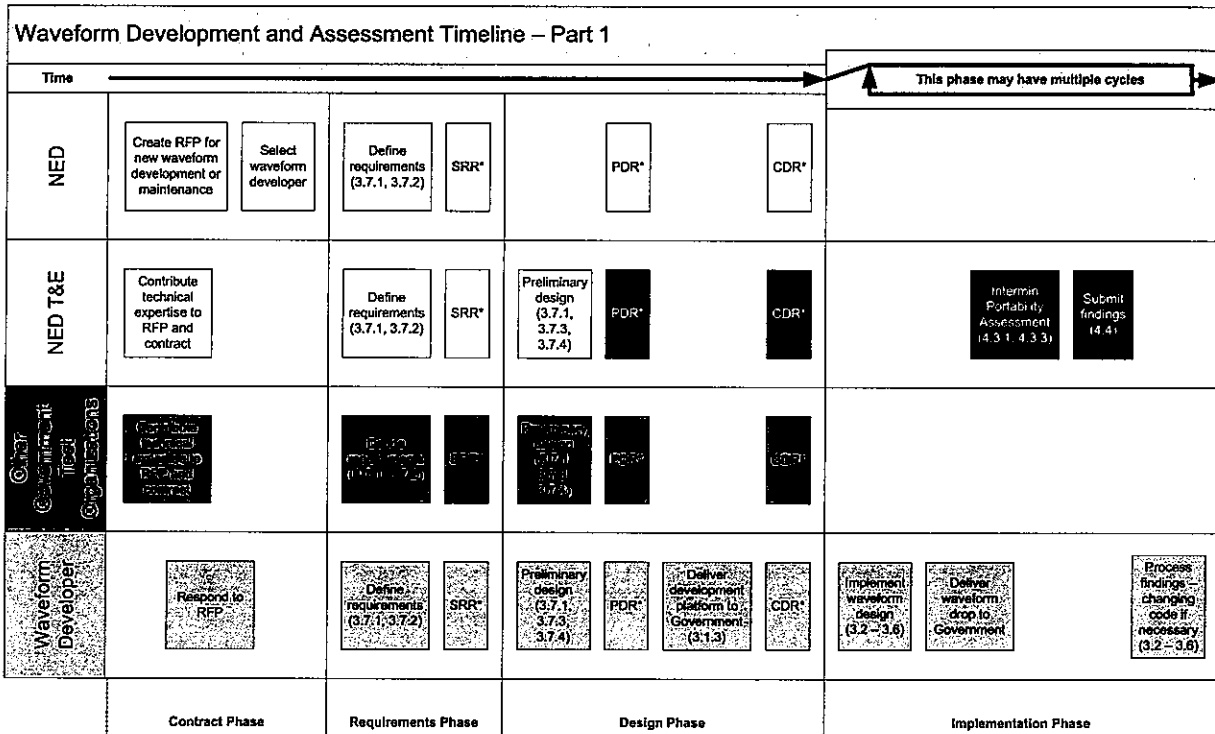


Figure 1-1. Waveform Timeline, Part 1

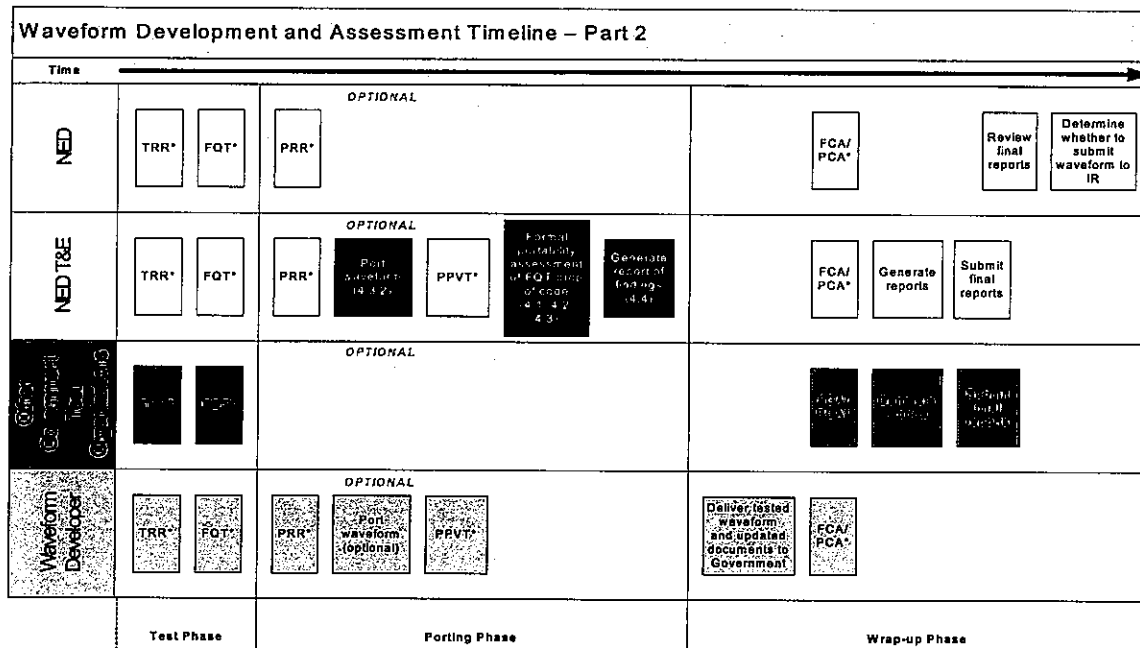


Figure 1-2. Waveform Timeline, Part 2

2 Referenced or Related Documents

This section lists all documents related to the creation, understanding, and use of this document. Subsequent revisions of these documents will apply.

- [1] JTRS NED T&E Waveform Portability Assessment Procedures, Version 1.1 DRAFT, 24 June 2008.
- [2] JPEO JTRS Charter, 14 October, 2005.
- [3] Joint Tactical Radio System (JTRS) Operational Requirements Document, Version 3.2.1, 28 August 2006.
- [4] Joint Tactical Radio System (JTRS) Standards, Joint Program Executive Office (JPEO) JTRS Software Communications Architecture Specification, Version 2.2.2, 15 May 2006.
- [5] Joint Tactical Radio System (JTRS) Standard Modem Hardware Abstraction Layer (MHAL) Application Program Interface (API), Version 2.12, 27 August 2009.
- [6] Joint Tactical Radio System (JTRS) Standard MHAL On Chip Bus Application Program Interface (API), Version 1.0.4, 27 August 2009.
- [7] JPEO JTRS Standards Standardization Plan, Version 1.9.1, 3 June 2009.
- [8] JTRS JPEO Software Standards, Version 1.2, 23 May 2007.
- [9] Stroustrup, Bjarne. *The C++ Programming Language*, Addison-Wesley, 2000.
- [10] International Standard ISO/IEC 14882:2003(E), Second edition, Programming languages — C++, American National Standards Institute, 2003.
- [11] International Standard ISO/IEC 9899:1999, Second edition, Programming languages — C, American National Standards Institute, 1999.
- [12] Object Management Group (OMG) C++ Language Mapping Specification, Version 1.1, June 2003.
- [13] Object Management Group (OMG) C Language Mapping Specification, June 1999.
- [14] IEEE 1076-2002, Standard VHDL Language Reference Manual, Institute of Electrical and Electronic Engineers, Piscataway, NJ, 2002.
- [15] IEEE 1164-1993 Standard Multi-value Logic System for VHDL Model Interoperability, Institute of Electrical and Electronic Engineers, Piscataway, NJ, 1993.
- [16] Cohen, Ben. *VHDL Coding Styles and Methodologies*, 2nd Edition, Kluwer Academic Pub. 1999.
- [17] NEDTE-WDS-DID-V1.0, Waveform Design Specification (WDS), 27 October 2009.
- [18] NEDTE-SRS-DID-V0.1 DRAFT, Software Requirements Specification (SRS).
- [19] NEDTE-SDD-DID-V0.1 DRAFT, Software Design Description (SDD).
- [20] NEDTE-IDD-DID-V1.0, Interface Design Description (IDD), 7 April 2009.
- [21] NEDTE-SVD-DID-V0.1 DRAFT, Software Version Description (SVD).
- [22] NEDTE-WPP-DID-V1.0, Waveform Porting Plan (WPP), V1.0, 13 January 2009.
- [23] NEDTE-WPR-DID-V1.0, Waveform Porting Report (WPR), v1.0, 31 August 2009.
- [24] NEDTE-SPS-DID-V0.1 DRAFT, Software Product Specification (SPS).

- [25]NEDTE-SOW-DID-V0.1 DRAFT, Statement of Work (SOW).
- [26]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [27]Fowler, Martin. *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Inc., 1999.
- [28]Riel, Arthur J. *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

3 Development Guidelines

As discussed in the Background section (1.2) of this document, the end product of a JTRS waveform development program is not a binary software application. The end product is a waveform consisting of a set of source files that can be hosted on a number of different platforms and supporting documentation necessary to enable a third-party entity to port and optimize the waveform for a specified platform. The documentation provided by the waveform developer assumes virtually equal importance to the overall success of the program as the software itself. This section provides guidance for waveform software development, as well as the supporting documentation.

Note: All JTRS waveforms must comply with the documents listed below. In the event of a conflict between one of the listed documents and these guidelines, the listed document will usually have precedence. The Government will determine any cases in which this document has precedence.

1. JTRS ORD [3] – The Operational Requirements Document. This describes the overall requirements that the JTRS program is expected to implement.
2. SCA Specification [4] – The Software Communications Architecture specification for the basic architecture used in all JTRS radios and waveforms.
3. JTRS MHAL Standard [5] – The JTRS Modern Hardware Abstraction Layer (MHAL) standard describing the interfaces between specialized hardware components [e.g., Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs)] used in JTRS radios and waveforms.
4. JTRS MHAL On Chip Bus Application Program Interface (API) [6] – An extension to the MHAL interface that describes a parallel interface between specialized hardware components (e.g., DSPs and FPGAs) used in JTRS radios and waveforms.
5. JPEO JTRS Standards Standardization Plan [7] – The JTRS program-wide plan for standardizing commonly used APIs between JTRS waveforms and radios.
6. JTRS JPEO Software Standards [8] – JTRS Test and Evaluation Laboratory (JTEL) standards for coding, documentation, and code delivery.

3.1 General Development Guidelines

Waveform developers should adhere to the following general development guidelines for waveform portability. Exceptions must be identified to the Government staff and the PAT prior to delivery of the PDR.

1. Logically subdivide the waveform into components that have well-defined interfaces between them. This design practice will facilitate hosting of these components on different Computational Element (CE) types [General Purpose Processor (GPP), DSP, or FPGA] and will support changing the type of element hosting a given component if necessary for porting to a new platform. These are multi-processor applications and should not have components tightly coupled to one another unless necessary to fulfill performance requirements. In such an instance, the developer must provide justification for any tightly coupled components.
2. In general, having one component, as defined in Section 3.1.5, per CE (e.g., one red GPP component, one black GPP component, one DSP component) is not an acceptable level of modularity in design unless the waveform is extraordinarily simple.
3. Provide as much unit test material (test vectors, documentation, etc.) as practical, along with high-level math models for any code related to signal processing. This is not a requirement for additional development, but for delivery of such materials already created and utilized by the

developers. Where possible, test vectors should be capable of being used to exercise individual waveform components in stand-alone form as an aid in unit testing.

4. Standards compliance is a necessity. The contract will identify the particular standards that apply to a given waveform. These standards will include compliance to the SCA and the JPEO API standards as verified by the JTEL test procedures.
5. Design the waveform so that, whenever practical, it has no requirements for Radio Services that are not available on all identified target platforms. If such a Radio Service is necessary, the service must be delivered as part of the original waveform and must follow the portability guidelines in this document. The new service must also be registered along with the associated API with the JTRS standards body.
6. Do not use contractor-developed tools or libraries unless provided to the Government with Government Purpose Rights (GPR) licenses, for inclusion into the IR along with the waveform. These items cannot have proprietary markings. This guideline should not be construed as prohibiting build scripts, which are necessary for the development process and which are considered a deliverable part of the waveform.
7. If source code generating tools are used, provide the generated source for evaluation, as if the waveform developer wrote it manually. Automatically generated code must be documented and models/artifacts provided along with the waveform source such that the WI can port it. The code-generating tool must also be commercially available for use by the WI. *Note: This does not include Common Object Request Broker Architecture (CORBA) stubs and skeletons generated by an IDL compiler. Note: In this context, "model" refers to a formal model than can be processed by a commercially available modeling tool such as Prismtech Spectra or Zeligsoft Component Enabler.*
8. Code and documentation must agree with respect to the details of waveform design and implementation.
9. Avoid basing the waveform design on strict timing expectations that could vary between platforms. The developer should document the rationale in the Software Design Description (SDD) if unable to conform to this guideline.
10. Establish performance characteristics (threshold and objective) for individual components to aid in understanding rehosting on different CEs. Include these characteristics in the delivered documentation.
11. Develop naming conventions for variables, functions, classes, and other software constructs, and use them consistently. Provide documentation on these notations – preferably in the Software Development Plan (SDP).
12. Use file naming conventions that comply with the rules set forth in the JTRS JPEO Software Standards [8]. Provide documentation on these notations – also preferably in the SDP.
13. Ensure proper memory partitioning and identify the contents of each partition. This is necessary for documentation of the waveform structure.
14. Build in as much debug capability as practical. When possible, this should be in the form of SCA logging as such code is SCA-compliant and should be designed and implemented so that it can be turned on and off, as needed. Log entries should include identification of the source file and line number whenever possible. In such cases where SCA logging is not practical, temporary debug code can be added. Include loopback capabilities to enable individual component testing. Debug code must either be removed prior to delivery to the Government or designed so that it can be removed from the executable using conditional compilation. If the code is removed altogether, the instrumented source should also be delivered to the Government for analysis.

3.1.1 Consideration of Intended Use of the Waveform

Above all else, the waveform must fulfill the capabilities for which it was developed. Under some circumstances, doing so will require compromises in portability, but that is preferable to a waveform that cannot be hosted on those platforms most suitable for its application. The JTRS goal of portability should always be considered during development, but some waveform design decisions may necessarily limit portability for reasons of performance, power limitations, etc. An example of this would be designing a waveform intended to function for an extended period on an internal battery power supply. Developers of such a waveform would need to be cognizant of the power consumption characteristics of the CEs available to them, and should allocate functionality accordingly. In this example, they might choose to allocate large parts of the waveform to a DSP. While a DSP-based design might be less portable in an absolute sense, the ability to port to the intended target platforms *and* fulfill the purpose of its development is improved.

3.1.2 Use of Third-Party Software

Use of third-party software, including freeware and shareware, is a common development practice that often returns benefits in schedule, cost, and functionality; however, these benefits can come at a cost of reduced portability. Many third-party software packages targeted for GPPs and DSPs deliver as object libraries that are linked in when building the executable. This makes it difficult, if not impossible to port the third-party components to other platforms. The main reason for this difficulty is the lack of visibility into the library implementation. Without the library source code, there is no way to verify the adherence to portability-enhancing programming practices. There is also no way to verify the compliance with applicable standards such as the SCA [4]. Finally, third-party software delivered as object code is always built for a specific processor, and usually built for a specific operating system, either of which is a significant barrier to portability. If the third-party software is available as source code, many of these objections may be put aside. The word “may” applies because the PAT will evaluate that source as if it were part of the waveform-specific source code.

Third-party code targeted at FPGAs is usually in the form of Intellectual Property (IP) cores provided by the component vendor. These will rarely work on different FPGA components and are discouraged for that reason. The same objections apply for third-party software not available as source code for the GPP and DSP components.

Another form of third-party software is legacy waveform code contained within an SCA-compliant wrapper. While such code may provide the necessary functionality, it is rarely optimized for the target platform(s) and can often contain barriers to portability.

The Government understands that there will be occasions where the benefits for inclusion of third-party software outweigh the disadvantages discussed herein. In such instances, the waveform developer must contact the PAT to discuss the exceptional circumstances that they feel make the use of such third-party software necessary. The PAT will evaluate the technical decisions and work with the waveform developer to choose the best option. Should the PAT and the developer agree to the inclusion of third-party software, the waveform documentation [Waveform Design Specification (WDS), SDDs, Interface Design Descriptions (IDDs), or Waveform Porting Plans (WPPs), as appropriate] must include a discussion of any portability issues related to the use of such software on the target platforms identified for this waveform.

3.1.3 Development Tools and Debug Code

The waveform developer shall provide a fully documented build process along with the identification and configuration of the development tools in the SDP. This permits reproduction of the executable during the assessment process and porting activities. This applies to all types of CEs.

In addition to all source code, the developer shall ensure that all debug code harnesses, project files, test benches, and simulations are included in the delivered software. These items are not part of the waveform source, but are valuable, and in some cases necessary, tools in the development and integrations processes.

3.1.4 Modeling

Commercially available software modeling systems exist to develop some or all of the waveform in a Platform Independent Model (PIM) for conversion to a Platform Specific Model (PSM). The value of tools such as MatLab and SimuLink when developing and porting the signal processing software has been widely recognized, and has become commonplace. For other aspects of the development process, the use of modeling tools varies between organizations.

The usual method of using such tools is to generate prototypes of the mathematical models of the signal processing algorithms and test them to ensure that the model produces the desired output under a variety of conditions. The developer then uses the model as a basis for the actual software and/or HDL. Some of the tools can generate the source code for the GPP (in C++), the DSP (in C), or the FPGA in [Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)]. Should the waveform developer use modeling tools to generate source, they should ensure that the source code conforms to the standard conventions of the appropriate language and must be compliant with the guidelines and standards outlined in this document.

Appendix D includes a more in-depth discussion of modeling techniques, tools, and standards.

3.1.4.1 PIM/PSM Modeling and Transformation

Designing a waveform as a PIM allows the developer to create a generic model that captures the necessary waveform functionality in a manner completely independent of the actual implementation of the radio. The available tools for transformation of a PIM into a PSM should create a working model for the chosen target platform without requiring changes to the original PIM. The reality is that there are only a few such tools available, and none of them can perform the entire transformation into a fully functional waveform optimized for the platform for which it is targeted. In addition, industry has not yet widely embraced the use of these tools in this manner. Still, the creation of a generic PIM as a part of waveform design and implementation has a great deal of value, even now, as a means of expressing the functionality of the waveform in a manner that another engineering team can easily grasp. The PIM also provides an insight into the motivations of the waveform designers. If the WI or the PAT has access to appropriate modeling tools, the PIM can be used to model waveform behavior, and can also be transformed into a representative PSM. The transformation process is where the waveform can be optimized for the target platform.

3.1.4.2 Signal Processing Code Modeling

The prototyping aspect of models is helpful in providing an understanding of the signal processing operation of the waveform. This understanding is valuable to the assessment team, as well as to a WI. As stated in the previous paragraph, the PIM provides a view of the structure of the waveform, as well as an insight into the motivations of the designers. The PSM provides a functioning view of the waveform on the development platform, but to be truly valuable, it must match the implementation in a bit-exact manner—that is, they must produce exactly the same output for the same input. This allows the WI, as well as the PAT to compare the performance of the ported waveform to the original reference model, isolating any inaccuracies induced during the port. Note that the developer shall deliver the models to the Government with the design documentation that includes detailed descriptions of the models and tools. This documentation shall also include the specific versions and configurations of the tools.

The developer should use simulation tools to perform system simulation at multiple levels. The simulations should include the test methodology, test benches, test vectors, verifiers, report generators, math models, and any other test material used. The developer should perform these tests and simulations at boundary conditions, as well as under normal operation, and should provide the Government with documentation of these simulation runs.

Some general guidelines describing what the waveform developer should include in models supplied to the Government are listed below.

1. The high-level math models of the signal processing software should agree with the delivered waveform implementation in a bit-exact manner (i.e., test vectors injected into both the delivered software and the model must yield the same output vector).
2. The waveform developer should provide as much unit test material (test vectors, documentation, etc.) as practical so that the PAT can execute the models and compare the output to that from the code.
3. FPGA simulations should include the test methodology, test benches, test vector stimulus files, verifiers, report generators, math models, and any other test material used. If these simulations are generated using elements that are considered company IP, the developer must provide some means for the simulation to be run without these elements. This is not likely to be possible, so the use of such elements is discouraged.

3.1.4.3 GPP Code Modeling

Modeling tools for components that are hosted on the GPP can be used for design only, or for the actual implementation. Artifacts from the design phase, such as Unified Modeling Language (UML) class diagrams, are usually included in the design documents.

Tool vendors have developed modeling tools that can take models created graphically and generate complex applications in third-generation languages such as C++. Such tools enable platform-independent development at a high level and can sometimes generate the platform-specific implementation. The model produced is useful both as a design artifact and potentially as a method of delivery for the waveform; however, this last possibility will require that the NED change its policy regarding waveform deliveries. Such a policy change is unlikely to occur until there have been more advances in modeling tools and more widespread acceptance of them by the industry. When machine-generated source code is included in a waveform delivery, it must conform to the same portability guidelines expressed in this document as manually developed code.

3.1.5 Waveform Architecture

One of the most important aspects of the overall waveform design success is its architecture—that is, how the individual components of the waveform are designed to implement certain functions required for the waveform, how these components are allocated to the target platform hardware, and how data and control move between the components. Note the use of the phrase “target platform hardware.” This is an instance where environmental conditions may outweigh abstract portability standards. In such instances, the desire for timely porting of a waveform between two similar platforms may make compromises in abstract portability acceptable. This section is intended to be a high-level discussion of architectural concepts and considerations rather than a complete and exhaustive treatise on these topics.

The grouping of functionality and the allocation of these grouped functions across platform computational elements is closely related to the concept of software modularity. It is safe to say that most programmers believe that they fully understand the concept of modularity and how to apply it; however, the degree to which any given waveform can be subdivided into separate components has proven to be open to interpretation. C/C++ programmers and VHDL developers sometimes use these terms in different manners, resulting in confusion between them. For this discussion, on the GPP, the term “component” is synonymous with an SCA *Resource*, and the term “module” applies to an individual C++ class. On a DSP, “component” will be used to describe major functional blocks, while “module” will refer to closely associated functions usually contained in one source file. Typically, a component, as defined here, will operate as an individual process on the GPP and DSP. On the FPGA, “component” better describes the individual building blocks contained within the top-level “entities”. These components are akin to C++ classes.

Consider the following four major software characteristics when allocating functionality to waveform components.

1. Cohesion – A measure of how strongly related and focused are the responsibilities of a single component.
2. Coupling – The degree to which each program component relies on each one of the other components.
3. Control flow complexity – A measure of how many conditional clauses exist in a software component.
4. Data/Control interfaces – The interconnections between components.

It is difficult to establish a set of “one size fits all” rules for waveform architecture, as each waveform’s requirements will differ; however, some general guidelines can be proposed.

1. Combine modules whose purposes are closely associated into components. Many modules may exist within a component.
2. Place modules into separate components if there is likelihood that they will need to be hosted on different CEs on one or more of the target platforms identified for this waveform.
3. The determination regarding component boundaries and component allocation to a platform CE should include consideration of:
 - a. Waveform instantiation time
 - b. Latency
 - c. Platform CE performance capabilities across all targeted platforms
 - d. Platform CE interconnection characteristics across all targeted platforms. This includes consideration of the performance capabilities of the hardware implementing these interconnections, as well as the software used to support them.
4. Avoid designing unnecessary or excessive SCA port connections, as these will increase waveform startup time and consume excess memory. Try refactoring software components to balance portability with performance.
5. Do not use SCA *Ports* for interconnections between modules within a GPP component.
6. C++ modules (classes) should follow rules of cohesion, encapsulation, coupling, etc., such as in *The C++ Programming Language* [9].
7. Design interrupt handlers to complete execution as quickly as possible. Use interrupt handlers to schedule work in threads rather than doing all of the work in the handler itself. Note that interrupt handlers are platform-specific and generally should not be a part of the waveform – especially in the GPP code. There is a great possibility that such code may be needed in the DSP code, but the necessary tight coupling to the platform means that it should only be a last resort.
8. Communication between components shall be compliant with the JTRS APIs. Additionally, use the interface as it was intended, e.g., do not use *configure()* when *query()* would be more appropriate.
9. Wrapping a non-portable legacy waveform in a thin, SCA-compatible layer does not produce a waveform that is any more portable than the original. Avoid this practice.
10. Avoid custom *SCA::Devices*. Such devices are not required to be SCA compliant with the SCA Application Environment Profile (AEP). The waveform should use those *SCA::Devices* provided

by the platform unless absolutely necessary. If a waveform developer feels that the development of a custom *SCA::Device* is necessary, they shall provide documentation that supports their position.

11. Avoid developing components with high cyclomatic complexity scores and complex test paths. Doing so supports maintainability and understandability, and thus enhances portability. Code with high cyclomatic complexity is only acceptable if no substitute design can be found.

3.1.6 Inline Documentation

Properly written software includes inline comments that provide the readers with additional information useful to gaining an understanding of the source code at which they are looking. This documentation is commonly referred to as source code comments. Some basic guidelines regarding source code comments appear in other sections of this document, but they are worth repeating and elaborating on here.

1. Source code comments shall be technically correct and current to the latest implementation of the code.
2. Inline comments shall agree with the design and implementation described in the waveform design documentation.
3. Both quantity and quality of inline comments are important to the understanding of the code. Adequate quantity is relatively easy to characterize.
 - a. Each source file shall have a comment header that, along with configuration management information and the Government data rights statement, includes a description of the function of the source code within the file. This description shall include the purpose of the source module, inputs and outputs, preconditions, postconditions, and any special notes particular to this source module. An example of such a note is an indication that the files contain platform-specific software that is likely to require changes during porting. Source file headers should conform to those rules outlined in the JTRS Software Standards [8].
 - b. Each function within a source file shall have a comment header that describes any input variables, output variables, the function algorithm, etc.
 - c. Any areas within a source file that the developer thinks may have portability implications should be clearly marked. A simple way to make these areas easily located is to add a specific tag in a header. This allows for keyword searches.
 - d. Developers should include particularly detailed comments in any areas where algorithm implementations may be obtuse or difficult to understand.
 - e. Logical blocks, such as *if-else* statements, *for* statements, *switch* statements, etc., should have a comment block above them that describes their purpose and how data and control flow processing takes place within them.
4. It is more difficult to characterize comment quality, but a few common sense rules are:
 - a. Comments should provide information describing the function and structure of the code in which it is located in a manner that is clear and readable.
 - b. Comments should be concise and should avoid requiring the reader routinely to refer to other documents.
 - c. Comments *must* be maintained so that they are in agreement with the code.

All of these guidelines apply to all CE types, including DSPs, FPGAs, and GPPs.

3.2 C/C++ Programming Practices

This section describes those recommended C and C++ programming practices that enhance application portability (Table 3-1) and those practices that are detrimental to portability (Table 3-2). Below are several standards that a waveform developer should follow.

- For C++, use the international standard *ISO/IEC 14882*, published by the American National Standards Institute (ANSI) as *ISO/IEC 14882:2003(E)* [10].
- For C, use the standard published by ANSI as *ISO/IEC 9899:1999, second edition 1999-12-01* [11].
- When using an Object Request Broker (ORB), C++ source code shall adhere to the C++ binding standard for CORBA 2.3 (<http://www.omg.org/docs/formal/03-06-03.pdf>) [12]. This applies to any processor type on which C++ is used.
- When using an ORB, C source code shall adhere to the C binding standard for CORBA 2.3 (<http://www.omg.org/docs/formal/99-07-35.pdf>) [13]. This applies to any processor type on which C is used.
- The use of the Standard Template Library (STL) and C++ Standard libraries, where appropriate, is preferred over the use of proprietary libraries. Where proprietary or other third-party libraries must be used, the delivery of the source code for these libraries to the Government with GPR is a requirement.

Bias in the following tables is toward the most portable programming practices. Some of the guidelines contained below may be less applicable as language standards evolve. The tables will evolve as necessary.

Table 3-1. Portable C/C++ Language Features

Recommended Programming Practice	Notes
Use of abstractions	Protects against changes propagating through code due to platform interface differences. Use of service and device abstractions hides platform interface details
Safe serialization and de-serialization (applies when SCA persistence mechanisms cannot be used)	Ensures that data is completely persisted and restored in a thread-safe manner, and that factors such as system and object state are taken into account
Segregate platform-dependent files from platform-independent files	Avoids confusion and allows for separation of build processes
Use careful naming procedures such as the use of prefixes	Helps assure uniqueness
Encapsulate message construction and parsing. Abstract the mechanics of data transportation and the characteristics of each participant of a data path.	This will insulate the application software from the transport layers and from changes to the makeup of transport connection points
<code>main()</code> must be in a C++ file	Prevents problems caused by the use of different startup sequences by the two languages. In addition to the C startup sequence, C++ calls the constructors for static objects.
Use the "common denominator" between members of C/C++ compiler families	Some development systems differ in how they handle some types between the C and C++ compilers
Put a new-line at end of file	Not having this breaks some compilers (and even some editors)
Always declare and define a default constructor	Some compilers simply will not work with classes that do not have these. To avoid inadvertent instantiation, declare the constructor as <code>private</code>
Declare local aggregates (such as	The only way to avoid loader errors in some compilers is to define

Recommended Programming Practice	Notes
arrays) uninitialized, and then initialize in code	such aggregates as static; however, doing so is not thread-safe. Initializing in code after the declaration is safe under all circumstances
Use virtual declaration on all subclass virtual member functions	Include virtual declarations in subclasses as well as the parents. This avoids warnings in some compilers and provides better documentation
Always declare a copy constructor and assignment operator	Avoids automatic generation of copy constructors by compiler; auto-generated ones are usually not optimal
Type scalar constants to avoid unexpected ambiguities	Avoids ambiguous function calls in overloaded functions
Use the .cpp filename extension	Supported by all C++ compilers, while .cc is not
Avoid implicit object construction. Declare constructors with a single argument as <code>explicit</code>	Constructors that take a single argument are also known as "implicit type conversion operators." They can be used to implicitly convert one data type to another ("implicitly" means "without the developer explicitly coding it"). Declaring the constructor as <code>explicit</code> prevents the compiler from using the constructor in this manner.
Use compiler control flags that enforce ANSI C/C++ practices	Encourages compliance with ANSI standards
Use <code>extern "C"</code> to prototype C functions that will be called from C++	Improves readability and avoids compiler problems

Implementing the practices outlined in Table 3-1 will improve the chances of a waveform being truly portable, but that improvement can be negated by other programming practices that inhibit portability. Table 3-2 below lists commonly used programming practices that are known to inhibit portability.

Table 3-2. Non-Portable C/C++ Language Features

Discouraged Programming Practice	Notes
Use of pragmas	Typically used to select language extensions or alignment changes; interpretation varies between compilers. As such, these are often platform-specific. If pragmas must be used, they must be thoroughly documented in the SDD
Use of bit-fields	Appear as structures with colons and may cause problems due to differences in endian-ness. As such, these are often platform-specific. If pragmas must be used, they must be thoroughly documented in the SDD
Use of unions	Alignment dependent. Also could cause problems with endian-ness. As such, these are often platform-specific. If pragmas must be used, they must be thoroughly documented in the SDD
Use of native types	Often compiler-specific; the identifier <code>int</code> is 16-bit or 32-bit, depending on compiler. Should always use <code>typedef INT_32</code> or <code>INT_16</code> , etc.
Floating point equality or in-equality	Floating point should always be compared to some epsilon. Variances between compilers may cause floating point math to produce very small residual values. Comparing such values to constants, such as 0, can produce unforeseen results. Example: Instead of <code>if (f == 0)</code> , a programmer should say <code>if (abs(f) < epsilon)</code> where <code>epsilon</code> is defined as a very small number
Pointer greater than or less than	Memory location-specific; the only pointer comparisons allowed are equality or in-equality

Discouraged Programming Practice	Notes
Use of Object Request Broker(ORB)-specific calls	Different CORBA ORBs often have differing calls for functions, such as initialization and exception handling; these can be masked by the use of macros
Use of "friend" mechanism in C++	This creates an instance of "overt coupling" wherein one class has access to the internal implementation of another, thereby, creating a dependency
Public data members in C++ classes	This can create an instance of "surreptitious coupling" wherein one class has access to the internal implementation of another, thereby creating a dependency. All data members should be accessible from outside the class only through "set" and "get" methods
Defining macros with trailing punctuation such as "," or ";	Macros should generally behave as a constant, a program statement, or a function. Macros that behave as program statements or functions should tolerate the addition of a semicolon at the end at the time of use. Many compilers will not tolerate two semicolons
C++ templates other than those defined in the STL	Not universally supported; implemented differently amongst compilers; results in code bloat. Templates often cannot be completely avoided, but programmers should be careful when using them and must thoroughly document their use
Embedded assembly language	Search for the keyword "asm"; allowed by the SCA [4], but still a portability risk; absolutely machine-specific and should be avoided in the GPP. Any use on the DSP should be thoroughly documented
Use of nested C-style comments	Not interpreted the same by all compilers
Nesting of namespaces beyond depth of two	Namespaces increase the size of linker symbols, whose length may be restricted on some platforms
Unnecessary top-level semicolons, i.e., do not put unnecessary semicolons in global scope	Some compilers will not accept unnecessary semicolons
Mixing of varargs and inlines	Not consistently implemented by all compilers
Use of nested classes	Implemented differently by some compilers depending on whether they follow the 1998 C++ standard or the 2003 C++ standard
Declaration of iterator variables inside for() statements	Prevents code compilation when using tools that lack support for "for-scoping"
Use of complex inline constructs	Unpredictable behavior from compiler to compiler; keep inlines short
Use of the mutable keyword	Leads to code that is difficult to debug
Definition of constants using simple names such as ERROR, FOREVER, and TRY that are the same as language key words	Such names may conflict with definitions in libraries used by the development system; such definitions should include something to identify the context in which they apply
Use of vendor-specific preprocessor directives	Not portable to other development environments
Absolute pathnames for included header files	Not likely to be portable to other development systems without change
Allocation of automatic variables whose size cannot be determined at compile time	Some compilers will allow automatic variables (e.g., buffers) to be allocated with a size based on the value of some other variable. This is not ANSI compliant and is not accepted by many compilers. Besides being non-portable, this is simply bad programming practice
using directives and using declarations in header files	Including a using directive in a header file will cause problems for anyone including the file. It can cause namespace collisions when

Discouraged Programming Practice	Notes
	included by other developers
Use of vendor-supplied macros	Macros supplied by compiler, ORB, and CF vendors are not likely to be portable
Use of C-style type casts in C++	Use of C++ static casts is preferred due to type safety and readability
Use of native C++ exception handling	Not portable to platforms that do not support native exceptions. Can adversely impact performance and increase code size

3.3 GPP Guidelines

3.3.1 GPP Performance

The waveform developer shall verify the extent of processor utilization in terms of Millions of Instructions per Second (MIPS) and memory requirements – both under peak processing load. The waveform shall be capable of execution on all target platforms with a specific minimum margin in terms of processing resource consumption on the least of these platforms. The specific margin value, and how it is determined will be defined by the Government in the waveform acquisition contract. MIPS allocations must include the GPP used determine processor utilization and the associated clock rate being used on the processor. The developer shall provide documentation for the capacity numbers, including their derivation, in the SDD. Note that there may be JTRS platform hardware requirements for GPP performance and memory margins that override this guideline. The developer and the Government should research the platforms selected for this waveform and identify any such issues.

3.3.2 POSIX Compliance

Waveform use of GPP Operating System (OS) services is constrained to the use of Portable Operating System Interface (POSIX) functions identified in the Application Environment Profile (AEP) defined in the SCA Specification [4]. The SCA standard prohibits the use of any other OS functions, as the use constitutes a serious threat to portability.

3.3.3 Use of JPEO Standard APIs

GPP interfaces between the waveform and the devices and services provided by the target platform shall be through APIs standardized by the JPEO in the API Standardization Plan [7]. Use of these APIs enhances the portability by ensuring that the waveforms and platforms speak the same language for these services.

These APIs will necessarily continue to evolve, and the JPEO may not yet have standardized some interfaces between the waveform and platform by the point of need in a particular waveform development effort. In such a situation, the waveform developer shall follow the guidelines set out in the Standards Standardization Plan [7] regarding the expansion of the existing standardized APIs and/or the development of new APIs for JPEO approval. When appropriate APIs do exist, the waveform shall conform to the API standards in place at the time of the contract award for that waveform development program.

3.3.4 Device Interface Abstractions

One of the characteristics of the SCA [4] is its reliance on abstractions of radio-specific devices to insulate the waveform-specific code from platform details. This is particularly common between waveform components hosted on the GPPs and hardware devices on the radio. The usual boundaries between waveform and platform software locate the abstraction for the device on the GPP and present interfaces for interaction with the waveform. The abstraction, implemented as an SCA *Device*, sits above a device- and platform-specific driver. The interface will hide details of the communication path between the waveform and the hardware device.

Refer to Figure 3-1 for an example of portable design. At the bottom of the figure are low-level devices connected to the GPP through either the FPGA or DSP. Below the waveform component is an SCA Device abstraction that implements the API for the device. This abstraction communicates to the device via the MHAL [5] (or MOCB [6]) layers on the GPP and the FPGA or DSP, as shown. Note that in no case does the waveform application ever interface with the low-level drivers. All communications occur through abstraction layers. As a result, waveform software communicating with the low-level device is insulated from changes to that device. The device interface software can be ported either to DSPs from another vendor, or hosted on a different device such as an FPGA.

Note: Figure 3-1 is a generic figure and does not represent the only possible configuration. Some configurations may include waveform components within the DSP and/or the FPGA.

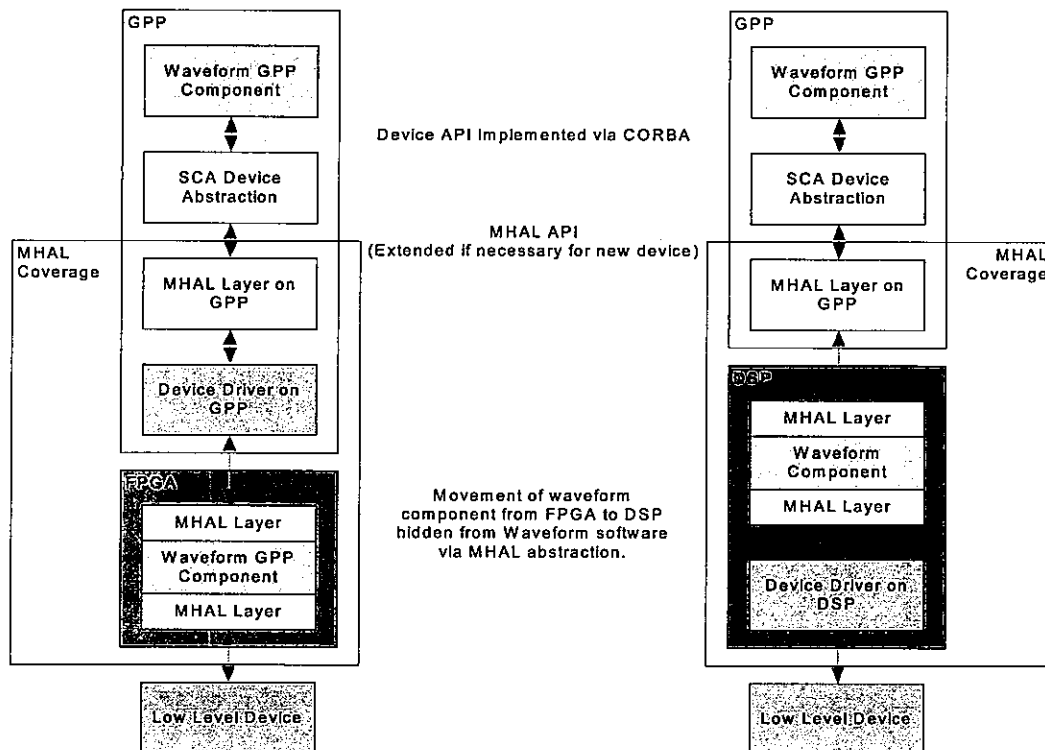


Figure 3-1. Example of Platform Abstraction

This picture changes when the platform uses an ORB on the DSP and/or FPGA. This change allows the abstraction to move closer to the actual device and removes the need for the use of MHAL APIs. This is relatively new technology and no designs reviewed by the Government to date include this capability.

3.4 DSP Guidelines

This section lists some guidelines for DSP development. NED T&E personnel and knowledgeable individuals in the software industry contributed to the development of these guidelines.

1. Verify the extent of processor utilization in terms of MIPS and memory requirements at peak processing load. The waveform shall be capable of execution on all platforms for which this waveform is targeted with a specific minimum margin in terms of processing resource consumption on the least of these platforms. The specific margin value, and how it is determined will be defined by the Government in the waveform acquisition contract. MIPS allocations must include the DSP used to determine processor utilization, the associated clock rate being used

on the processor, any internal clock multipliers, or the core clock rate used. Provide documentation for these numbers, including their derivation, in the SDD. Note that there may be JTRS platform hardware requirements for DSP performance and memory margins that override this guideline. The developer and the Government should research the target platforms identified for this waveform and identify any such issues.

2. If CORBA connectivity does not exist in the DSP, ensure that all interfaces to other CEs and remote devices comply with either the JTRS MHAL [5] or MOCB [6] standard.
3. Limit use of processor-specific OS functions. If use of such functions is mandatory, abstract them through local functions. Limit use to basic functions that other DSP operating systems are likely to support. The developer should research the target platforms identified for this waveform and document any portability issues related to differences in DSP OSs. This documentation can be located in the SDD or a WPP, as appropriate.
4. Avoid assembly language unless it is necessary to meet performance requirements. The waveform developer shall provide justification for the use of assembly language. In the SDD, fully document the purpose and execution of any embedded assembly language and include a version of the algorithm written in a portable, high-level language.
5. The waveform developer shall put in place a set of DSP coding standards requiring that they abide by all language rules, provide a common look and feel to the code across all modules, and be readable and maintainable.
6. Exercise caution in the use of mathematical operations where the C++ standard [10] and the C standard [11] are ambiguous and permit the developer to be "implementation-specific", such as in the case with the modulo/remainder operation on negative numbers in the C++ standard. The developer shall implement the code in a way that does not rely on the operation of a specific compiler in these cases.
7. Exercise caution when using chip support libraries or other similar processor-specific APIs or functions within waveform DSP code. Whenever possible, avoid functions that are not likely to be commonly available across multiple component's support libraries. The waveform developer shall fully document the purpose and execution of any processor-specific (i.e., chip support) APIs or functions in the SDD.

3.5 FPGA Guidelines

This section lists some guidelines for FPGA development. NED T&E personnel and knowledgeable individuals in the software industry contributed to the development of these guidelines.

Note: The Government considers any VHDL written for the waveform and required for its operation to be software, and to be a deliverable along with the GPP and DSP source.

1. Source code shall conform to active VHDL standards, 1076 [14] and 1164 [15], set forth by the Institute of Electrical and Electronics Engineers (IEEE).
2. Ensure that all interfaces to other processors and remote devices comply with either the JTRS MHAL [5] or MOCB [6] standard.
3. When an IP core is used, documentation describing all input parameters used to generate the IP core shall be provided in the SDD. In general, the more generic IP cores (e.g., simple adders, multipliers) are more acceptable as the WI can reasonably expect them to be available on devices other than the original target. If the IP core in question is known to be available on components from other FPGA manufacturers, its use is acceptable.

4. The VHDL system architecture should use modular components that provide a logical scope of functionality. Each entity should have a clear and defined role in providing functionality to the system.
5. Verify the extent of FPGA resource utilization at peak processing load. Ensure that there is sufficient margin (at least 25%) remaining. Provide documentation for these numbers, including their derivation, in the SDD. Note that there may be JTRS platform hardware requirements for FPGA performance and memory margins that override this specific guideline. The developer and the Government should research the target platforms identified for this waveform and identify any such issues.

Note: NED T&E engineers compared and analyzed the differences between specific JTR platforms, and determined that this guideline provides reasonable room for growth. JTRS product lines may have different margin requirements. Where those requirements differ from the 25% margin discussed here, the PAT will use the higher of the two numbers.

6. Abstract or eliminate the use of off-chip resources. Off-chip resources include anything from memory, Analog to Digital Converter (ADC), or even a serial interface to a DSP or GPP. The idea is to create a component driver for each off-chip resource to abstract the component from the waveform or core FPGA processing using MHAL or MOCB. This approach allows porting of the code to a new platform, changing only the individual components, and leaving the core unmodified. Still, it is best to eliminate the need for such off-chip resources, if practical, such as those resources that may not exist on other platforms. If system performance requirements cannot be satisfied without tightly coupling the waveform FPGA code to the platform, the developer must provide the rationale for this, and must thoroughly document the structure and functionality of the interface to the off-chip resource in the SDD.
7. Use of vendor specific primitives should be avoided unless absolutely necessary. If such primitives must be used, the developer must document them thoroughly and provide rationale for their use in the SDD.
8. Retain simulation portability by avoiding the use of simulation tool-specific simulation control language or APIs.
9. Minimize the number of clock domains to reduce/eliminate hardware dependence on multiple clocks; this practice will reduce design size, complexity, and resources and lower power consumption. Documentation of the clock domains must be included in either the SDD, or a separate firmware design document.
10. Use generics as component parameters to help eliminate platform-specific buried constants and optimize code.
11. Use 'type' objects to enhance the ability to use generics and simplify code.
12. The waveform developer shall put in place a set of VHDL coding standards requiring that they abide by all VHDL language rules, provide a common look and feel to the code across all modules, be readable and maintainable, and avoid obsolete VHDL constructs. Following the guidelines in VHDL coding style guides, such as *VHDL Coding Styles and Methodologies* [16], will achieve this.
13. Any software hosted on embedded GPP or DSP cores shall be implemented following the guidelines in Sections 3.3 and 3.4 above, as appropriate.

3.6 Domain Profile Guidelines

The Domain Profile is a collection of eXtensible Modeling Language (XML) files that the radio uses as a list of instructions of how to load, configure, and start up the waveform. The list below contains some guidelines for Domain Profile development. NED T&E personnel and knowledgeable individuals in the software industry contributed to these guidelines.

1. Adhere to Appendix D of the SCA [4] for format and content of the Domain Profile XML.
2. There are commercially available Domain Profile editing tools oriented toward the SDR industry. Using one of these tools will aid in the development of error-free XML. If such tools are used, supply the resulting model to the Government. The Government already requires that the developer submit XML files.
3. Use care when assigning Universally Unique Identifiers (UUIDs) as the assignment process can be error prone. Use either a script to keep track of the UUIDs and then run it to input the correct UUID into each field, or alternatively use a graphical Domain Profile editing tool.
4. Validate XML files in accordance with SCA Document Type Definition (DTD) files.
5. Define all waveform configurations and properties inside the XML Properties Files (PRF).
6. Define memory requirements for each component in the Software Package Descriptor (SPD) file(s).
7. Provide a properly formatted and written Software Component Descriptor (SCD) file(s). Even though the SCD is not required by all SCA Core Frameworks (CF), it is still good documentation for understanding the waveform.

3.7 Documentation Guidelines

Waveform documentation serves three important functions related to enhancing waveform portability. First, it provides all parties with a clear understanding of the expectations from each respective party. Second, it provides anyone involved in porting or assessing the waveform a clear understanding of how the waveform is structured and how it functions. Third, it identifies and discusses issues related to porting to a specific target platform. The SOW, the Waveform Design Specification (WDS), and the Software Requirements Specification (SRS) embody the first function. The SOW, in particular, will capture the contract-related issues. The WDS, SRS, SDD, and IDD(s) embody the second function. The WPP and Waveform Porting Report (WPR) embody the third, along with contributions from the design documents.

Note: Appendix C contains guidelines for Government development of the SOW.

Assessment of the deliverable documents will go beyond simply determining whether each document fulfills the requirements of the appropriate Data Item Description (DID), although meeting those requirements is still necessary. The assessment team will review the documents for readability, for how well it conveys the information necessary to understand the structure and function of the waveform, and for what implementation changes are necessary to port the waveform. The PAT expects these documents to be consistent with the delivery of the software in which they are included. This applies to interim deliveries, as well as the final delivery.

Several of the documents discussed in this section benefit from the inclusion of UML diagrams. State diagrams, sequence diagrams, and timing diagrams (added in UML 2) are particularly good at conveying information regarding program flow and module interactions. Class diagrams, component diagrams, and object (also known as instance) diagrams are useful in describing program structure. All of these are more useful in the WDS, SDD, and IDD documents as these are the ones that provide the most information on the waveform design and implementation.

Documentation deliverables include everything necessary to recreate the design and its motivation without the need to consult the waveform designer. The specific documents detailed below are the minimum acceptable to the Government.

3.7.1 Waveform Design Specification

The WDS is a top-level design document describing how waveform functional, operational, and performance requirements translate into hardware and software specifications. The document specifies requirements for the integrated system of waveform and platform—such as tuning range and accuracy, receiver sensitivity, dynamic range, timing requirements, Communications Security (COMSEC) modes, Transmission Security (TRANSEC) latency, and Over-the-Air (OTA) protocols—to hardware and software sub-systems. Perform this allocation and underlying analysis for all intended target platforms. Clearly identify constraints and restrictions that reduce the set of operational requirements achievable in any intended environment. In most waveform acquisitions, the WDS is the primary document from which requirements are derived for inclusion in the SRS. Some acquisitions reverse the process and generate the WDS from the SRS. In either case, the WDS provides the context necessary for the PAT and a WI to understand the rationale for the requirements. The WDS *does not* capture design details.

In some instances, it may be necessary to assign some waveform functionality to hardware rather than software in order to meet system requirements. The waveform developer should explicitly identify and explain instances where a system requirement cannot be satisfied without an adverse impact on portability.

The WDS shall comply with the following guidelines and shall conform to the WDS DID, NEDTE-WDS-DID-V0.1 DRAFT [17].

1. List any requirements placed on the target platform(s) by the waveform.
2. Describe the functionality of the overall system, such that a reader not familiar with this particular waveform can understand it clearly.
3. Clearly define all protocols and data translations in a manner that focuses on the necessary functionality and is not implementation-specific. Include all timing requirements, packet formats, etc., that are specific to the protocol, but not to the waveform's implementation of it.
4. Clearly identify constraints that impede test coverage.
5. Clearly separate the waveform components from the platform components required by the waveform.
6. Identify the types (GPP, DSP, and FPGA) of CEs required by the waveform and what waveform functions will be hosted on what CEs.
7. Describe the technologies the developer intends to use for waveform component developments; including programming languages, code generators, modeling tools.
8. Identify and discuss those properties of the waveform and/or platform constraints that are likely to impede portable design.
9. Identify all of the programmatic requirements applicable to the waveform, such as compliance with the SCA [4] and adherence to these portability guidelines.

3.7.2 Software Requirements Specification

The SRS lists the requirements that are usually derived from the WDS. The SRS for a portable waveform should state waveform requirements for all intended environments. The SRS shall comply with the following guidelines and shall conform to the SRS DID, NEDTE-SRS-DID-V0.1 DRAFT [18].

1. Require that the waveform components must be able to fit within the device capacity, such as memory and gates, of the hosting CEs for all of the intended target platforms.
2. Indicate applicability across all intended target platforms for each requirement.
3. Clearly separate software requirements allocated to the waveform from those that the waveform software requires of the platform.
4. List the timing requirements for each waveform component.
5. Provide the requirements traceability and analysis used in requirements flowdown.

3.7.3 Software Design Description

The SDD describes the design of a Computer Software Configuration Item (CSCI). It describes the CSCI-wide design decisions, the CSCI architectural design, and the detailed design needed to implement the software.

In a typical development, design documentation is available long before software source code. Many software development process manuals indicate that the majority of the life cycle leading up to test and delivery is dedicated to the design phase. For portable software, it is essential that attributes of the design that give rise to portability are manifest throughout the design documentation. Per the SDD DID, [19], elements of the SDD are a description of the CSCI-wide design decisions and architectural design, and a decomposition of the software design. For JTRS waveforms, it should also include a description of SCA interfaces, APIs, communication protocols and *Devices*, and a description of how the design supports all intended target platforms. The SDD shall comply with the following guidelines and shall conform to the SDD DID, NEDTE-SDD-DID-V0.1 DRAFT [19].

1. Provide an overall view of the structure of the waveform, including visibility into the DSP and FPGA components. This must include illustration of the allocation of waveform resources and components to CEs, and the connections between them.
2. Discussion of approaches considered and rationale for decisions made.
3. Include a complete end-to-end description of user data and control flow through the system, *including details on the translations that occur along the way*. The information highlighted in bold italics is particularly important.
4. Identify the distinct adaptation components that allow the waveform to overcome different hardware device interfaces among target platforms. Figure 3-1 shows an example of adaptation components used in concert with the MHAL for abstracting devices.
5. Contain the descriptions of the MHAL or MOCB abstractions used for all non-GPP computational nodes not already provided by the target platform as radio services or radio devices, e.g., FPGA, DSP, Antenna(s), Low Noise Amplifier (LNA), Automatic Gain Control (AGC), ADC, serial and audio ports, Digital to Analog Converter (DAC). Refer to the MHAL Standard [5] or the MOCB Standard [6] for details regarding the use of these interfaces for such devices.
6. Clearly differentiate between waveform components (*Resources*) and platform components (*Devices*).
7. Identify all expected “uses” and “provides” port connections between waveform components and between waveform components and the platform.
8. Identify any non-CORBA connections between waveform components and the platform that do not comply with either the MHLA or MOCB standard. If any such connections exist, the SDD should provide a rationale for them.

9. Describe, in detail, any non-CORBA and non-MHAL/MOCB interfaces. Include protocol definitions.
10. Provide justification for implementation of waveform components in an FPGA that could be implemented in the DSP or GPP. Generally, a GPP or DSP implementation of a component is more portable than an FPGA representation of the same component.
11. Clearly describe hardware dependencies, including memory usage, MIPS requirements, latency limits for any platform-related APIs or signaling, specialized interprocessor interfacing, specialized system clocking mechanisms. Provide to the Government the rationale behind these dependencies.
12. Point out areas that are likely to have portability implications and discuss those implications.
13. Define all configuration parameters, their ranges, and their default values.
14. Contain DSP and FPGA design documentation that includes the following.
 - a. Description of the data and control signal flow paths, including sampling rates at all internal and external interfaces, number of samples used to represent the signal and numerical representation of the signal at all internal and external interfaces, associated buffer/memory usage, and any control interfaces to the signal path
 - b. High level architecture design diagram
 - c. Detailed diagrams and descriptions of individual components, including a description of the clock frequencies used. If multiple frequencies are used, the documentation must include a description of the clock boundaries
 - d. Detailed descriptions of all state machines
 - e. Detailed descriptions of adaptive types of algorithms and analysis behind adaptive loop parameters used and decisions reached
 - f. Schematics of VHDL designs
 - g. Detailed descriptions of the method of synchronization, including, but not limited to, frame-level synchronization, slot-level synchronization, bit-level synchronization, and interprocessor synchronization
 - h. Indication of where component vendor-specific code was used and why. This includes descriptions of the functionality of such code, as well as the method(s) and parameter(s) for its generation
 - i. Detailed register definitions
 - j. Detailed description of timing considerations and use of interrupts
 - k. Detailed description of the timing of any other critical events
 - l. Interface timing diagrams
 - m. Static timing analysis
 - n. File hierarchy and/or diagram indicating boundary between waveform and platform components; files containing platform functionality should be distinct from those with waveform functionality
 - o. Design hierarchy showing section number in WDS that each item implements
 - p. Wrapper/Interface descriptions

3.7.4 Interface Design Descriptions

The waveform developer should identify all devices, services, interfaces, libraries, and macros considered external to the waveform (i.e., provided by the platform) and should describe, in detail, the interactions between these external entities and the waveform. The waveform developer will create one or more IDD's to provide these descriptions. These IDD's should correspond to APIs approved by the Interface Control Working Group (ICWG), per the Standards Standardization Plan [7].

The PAT considers waveform-specific SCA *Device* abstractions that are instantiated with the waveform to be part of the waveform and to present significant portability issues. Despite the portability risk, some waveform developers have chosen to include them. This practice is discouraged.

The assessment team will review the IDD's both for readability and for how well they convey the overall structure, implementation, and functionality of the waveform interfaces. The waveform developer should adhere to the following API guidelines and shall conform to the IDD DID, NEDTE-IDD-DID-V0.1 DRAFT [20].

1. The JPEO JTRS mandated use of the ICWG-approved APIs for waveform/platform interfaces; the Standards Standardization Plan [7] lists the approved APIs. If an existing API is not adequate, follow the rules defined in the Standardization Plan for expanding an existing API or creating a new one.
2. Non-CORBA waveform interfaces should conform to the MHAL [5] and/or MOCB [6] standards.
3. In accordance with the SRS, verify for each porting target that all waveform operations required by each waveform SCA *uses* port can be satisfied by the available operations on the platform services SCA *provides* ports. All waveform required *uses* operations should have a corresponding platform or external device SCA *provides* API.
4. For GPP waveform components, every interaction with the DSP/FPGA should use either the *MHALDevice* or *MOCBDevice* interface. The API documents must address the impact of whichever of these interfaces is used on latency of command and user data.
5. The IDD should include the IDL for all CORBA interfaces.
6. The IDD should include the sequence of commands necessary for the entities on both sides of an interface to reach their operational state. This will aid in independent testing of waveform components.

3.7.5 Software Version Description

The Software Version Description (SVD) serves the dual purpose of a shipping list and set of build instructions for a specific waveform software delivery. In compliance with the SVD DID, NEDTE-SVD-DID-V0.1 DRAFT [21], the SVD shall:

1. Contain a complete inventory of the files used to build the waveform. This includes all GPP, DSP, and FPGA source. It also includes all IDL files, as well as any other files used in the build process. This inventory must also include the size of each file and their version numbers.
2. List all XML files necessary to deploy the waveform.
3. Indicate which Software Assembly Descriptor (SAD) file is used for waveform instantiation.
4. Completely describe the target environment on which the release of the waveform being assessed is intended to function. The PAT must be able to duplicate this environment in order to determine that it has built a functional waveform.
5. List all documents relevant to this version of the software. This must include version and release dates for each.

6. Include a detailed description of the build environment, including version numbers for each tool used.
7. Provide instructions for building the waveform application that are complete and correct. The PAT will use them to attempt a build of the waveform.
8. Contain a list of problems or limitations that apply to this software release.
9. List any and all dependencies on the OE so that the PAT and a WI will know what OE components (ORB stubs and skeletons, header files, etc.) are required to build the waveform.

3.7.6 Waveform Porting Plan

The WPP contains information that describes the steps that a WI must take in order to rehost the waveform on a platform other than that on which it was developed. The WPP shall include a comparison between the original development platform and the target for which they write the document. This comparison will identify issues that could impede the port, allowing for their mitigation, if possible. The WPP shall comply with the following guidelines, as well as with NEDTE-WPP-DID-V0.1 [22].

1. Describe a waveform structure matching the one described in the WDS, SDD, and IDD.
2. Include sizing and timing analysis for both the original and target platforms for the waveform components hosted on the GPP(s), DSP(s), and FPGA(s).
3. Include gap analysis for those cases where the original and target platforms have different hardware.
4. Specify what changes are required for each waveform or platform component.
5. Address any issues where the target platform may not have sufficient resources to support the waveform. If such issues exist, the WPP should provide a mitigation strategy.
6. Identify the requirements that the PPVT will verify.
7. Describe how the PPVT will be performed.
8. Identify the resources required to port the waveform; include personnel, laboratory facilities, equipment, and tools [both Commercial-off-the-Shelf (COTS) and vendor-developed].
9. Identify any licensing issues regarding waveform components or those platform components that may be present in the development platform but not in the target platform, and that are required by the waveform.
10. Provide a detailed schedule of the port.

If there are any conflicts between these guidelines and the DID cited above, the DID has precedence.

3.7.7 Waveform Porting Report

The WPR describes how the waveform port proceeded, including discussion of any unforeseen issues that may have come up, how close the WPP was to the actual port, and the results of the PPVT. The WI should include in the WPR metrics that describe the number of lines of code that they reused, changed, and added during the port. This information will quantify the amount of rework required on the waveform. The WPR shall comply with the following guidelines as well as with NEDTE-WPR-DID-V1.0 [23].

1. Describe, in detail, any required changes not identified in the WPP.
2. Describe, in detail, any changes identified in the WPP that were not required.
3. Describe the functionality of any new waveform or platform components developed as a part of the port.

4. Describe, in detail, any structural changes required as a part of the port.
5. List the results of the PPVT, providing explanation for those tests that failed.
6. Provide details of the actual schedule of the port, describing any variances from the schedule included in the WPP.
7. List any unplanned resources required for the port.
8. List tools and/or methods used to generate the metric included in the WPR. This should preclude any confusion due to differences in measurement methods.

If there are any conflicts between these guidelines and the DID cited above, the DID has precedence.

4 Portability Assessment Background

There are four main functional areas tested on each JTRS waveform: SCA [4] compliance (performed by the JTEL), Performance, and Portability (both performed by NED T&E). Each organization involved will perform either tests or assessments to determine if the waveform is compliant in its area. This document focuses on Waveform Portability, detailing the guidelines developed to ensure that the waveform complies with JTRS portability objectives. The WPAP [1] uses these same guidelines as a basis for the assessment criteria. The following sections summarize the assessment process criteria and application.

4.1 Portability Qualities

Section 1.2.4 lists the qualities that the PAT must assess to determine the portability of a waveform. The rationale for each of these qualities appears in Table 4-1 below. The rightmost column indicates some of the questions that the PAT must ask to determine if the waveform embodies the quality described in the leftmost column. To make these determinations, the PAT must consider three major aspects of the waveform: documentation, architecture, and implementation.

Table 4-1. Portability Quality Rationale

Portability Quality	Rationale	Determination
Traceability	To be considered portable to a given platform, a waveform must implement its full set of requirements on not only the original platform, but the new target platform(s) as well. Only requirements listed as optional in the SOW may be omitted from the port. The PAT analyzes the design to ensure that it implements the waveform's requirements without requiring hardware or software components that are unavailable on the waveform's respective target platforms.	Does the documentation supplied provide proof of complete requirements coverage? Does the documentation include traceability through design and test?
Extensibility	Porting to a new platform inevitably involves adapting code to new interfaces, new devices, and new functions. The PAT considers the modularity of design and degree of difficulty associated with modification or extension of a waveform as part of the portability assessment. In some cases, a platform-specific implementation may be acceptable, but design details must be provided for any redesign and optimization.	Does the documentation clearly describe the waveform so that the WI can understand its architecture ? Is the described architecture one that enables easy modification and/or extension of the waveform? Does the developer's implementation avoid platform-specific interfaces? Does the implementation abstract platform interfaces?
Efficiency	Often a waveform must be ported to a platform with different or more stringent resource constraints than the original platform. The PAT evaluates the memory, throughput, latency, and bandwidth requirements of the waveform to ensure that they fall within the constraints of targeted or potential JTRS form factors.	Is the waveform designed with an architecture that maximizes efficiency for resource-constrained platforms, or does it appear to assume unlimited resources? Did the developer's implementation maximize performance of the compilers and promote efficient use of system resources?
Testability	Waveforms typically require a fair amount of debugging during a porting	Does the waveform architecture provide for testing of individual modules?

Portability Quality	Rationale	Determination
	effort and it is important that the waveform provide a means of verification of its internal functionality. The PAT will take into consideration the amount of support included for debug, tracing, inspection, and verification of waveform functions.	Does the architecture allow injection of test data and visibility of the results? Did the developer's implementation include debug and tracing capabilities?
Understandable Design	It is impossible for the PAT to assess, and the WI to port, something that they do not understand. It is the waveform developer's responsibility to provide sufficient documentation to provide both the PAT and the WI with sufficient details. The documentation must be clearly presented so that they gain the necessary level of competency with the waveform software.	Does the documentation clearly and concisely describe the waveform in a manner that is easily understood? Does the waveform architecture follow best practices for qualities such as modularity, etc.? Does the implementation include good in-line documentation that clearly describes the implementation of the design?
Maturity	Extensive testing is necessary to verify that all required functionality is still present in a ported waveform. To make this achievable, the waveform must already be thoroughly tested and understood on its original platform so any differences in behavior on the new platform can be identified. Any waveform delivery to be assessed for portability must be complete and functional on its original platform, and a complete set of test procedures – and errata, if any – must be supplied in the delivery. This includes test vectors.	Does the test documentation present a clear picture of a waveform that performs as required on its development platform? Did the developer's coding practices include the creation of test vectors, and were those vectors provided in the waveform delivery?
Ease and Degree of Work Necessary to Port	This is a measurement of the waveform's portability generated during the assessment process. The Government's goal is to port waveforms in less time and for lower cost than required for new development.	Does the documentation clearly indicate those areas such as interfaces that are particularly important relative to portability? Does the architecture embody the principles described in this document for enabling portability? Does the implementation follow the guidelines described herein that enable portability?

4.2 Assessment Methodology

All waveforms are different, and the steps necessary to assess their portability will vary. Even so, activity will take place for each waveform of the three areas described in the following paragraphs. Refer to the WPAP [1] for a detailed description of the assessment process.

4.2.1 Laboratory Analysis

While a skilled software engineer can learn a great deal about the portability of a waveform application by a detailed analysis of the source code and documentation, there are tools and techniques for a laboratory that will more efficiently locate and understand waveform portability characteristics. One of the first steps NED T&E will take is to build the waveform on a copy of the original development environment to establish a baseline. Next, a simple step proven very effective is to attempt to build the waveform using tools that are different from those with which it was originally developed, and targeting the waveform GPP software at a different Real-Time Operating Systems (RTOS) and ORB. This step will immediately point out the use of

development tool-specific programming practices. This practice goes beyond work done for a port of the waveform, because it is quite possible that the target platform and the development platform will use the same components (RTOS, ORB, etc.) and will use many of the same tools (compilers, etc.).

Also valuable are simulation and modeling tools used to determine the performance envelope of software components. These tools will be helpful to determine waveform component portability to a system that has less capacity than the development platform in the appropriate CE. Adding instrumentation to the code will provide the assessment team with similarly useful information for analyzing timing critical areas in the waveform.

The assessment procedures list a number of particular waveform characteristics that the PAT will assess in the lab. These characteristics will result in the generation of a portion of the overall rating given to the waveform.

4.2.2 Waveform Porting

The most effective way to judge the portability of a JTRS waveform is to port it to one of the target platforms identified in the Contract Phase of the waveform acquisition process. Most waveforms will have some portion of their software ported as part of the assessment process. In those cases where porting only portions of the waveform is practical, efforts will be focused on those areas that appear through static analysis to have the most potential for problems with portability. The PAT will compare those waveform components ported during the assessment to the original version of the component for changes in functionality and performance. This tactic uses the original development environment provided by the waveform developer, and the development environment for the target platform, as well as use of the target platform itself, when practical. Such a side-by-side comparison identifies any areas in which the component's performance changes due to the port.

4.2.3 Static Assessment

The static portion of the assessment process involves the review and assessment of the waveform source code and provided documentation. In this process, the PAT derives an objective measurement of portability from the comparison of certain waveform characteristics to applicable ideals, which results in the remainder of the portability rating generated for the waveform. The PAT also reviews the documentation for the presence of necessary information and its readability.

4.3 Assessment Results

For the final, formal assessment, the PAT will develop a Waveform Portability Assessment Report (WPAP) to document the findings of the assessment. It will include a discussion of portability issues related to all target platforms identified for this waveform.

Assessments of interim drops will generate lists of findings that the PAT will convey to the developer for corrective action. Refer to the WPAP [1] for details on how the waveform is assessed and how the final portability rating is generated.

Appendix A Acronym List

The following list includes acronyms and abbreviations used in this document.

ADC	Analog to Digital Converter
AEP	Application Environment Profile
AGC	Automatic Gain Control
ANSI	American National Standards Institute
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
BFM	Bus Functional Model
CDR	Critical Design Review
CE	Computational Element
CEA	Cryptographic Equipment Application
CF	Core Framework
COMSEC	COMmunications SECurity
CORBA	Common Object Request Broker Architecture
COTS	Commercial-off-the-Shelf
CSCI	Computer Software Configuration Item
CSMA	Carrier Sense Multiple Access
DAC	Digital to Analog Converter
DID	Data Item Description
DoD	Department of Defense
DSL	Domain Specific Language
DSP	Digital Signal Processor
DTD	Document Type Definition
FPGA	Field Programmable Gate Array
FQT	Formal Qualification Test
GPP	General Purpose Processor
GPR	Government Purpose Rights
HDL	Hardware Description Language
HDVL	Hardware Description and Verification Language
I/O	Input/Output
IA	Information Assurance
ICWG	Interface Control Working Group
IDD	Interface Design Description
IDE	Integrated Development Environment
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IF	Intermediate Frequency
INFOSEC	INFORMATION SECurity
IP	Internet Protocol or Intellectual Property
IR	Information Repository
ISO	International Organization for Standardization
ISS	Instruction Set Simulators
IWPA	Interim Waveform Portability Assessment
JPEO	Joint Program Executive Office
JTEL	JTRS Test and Evaluation Laboratory

JTR	Joint Tactical Radio
JTRS	Joint Tactical Radio System
LNA	Low Noise Amplifier
MDA	Model Driven Architecture
MDD	Model Driven Development
MHAL	Modem Hardware Abstraction Layer
MIPS	Million Instructions Per Second
MOCB	MHAL On-Chip Bus
MOF	Meta-Object Facility
NED	Network Enterprise Domain
NED T&E	Network Enterprise Domain Test and Evaluation
OCL	Object Constraint Language
OE	Operational Environment
OMG	Object Management Group
OO	Object-Oriented
ORB	Object Request Broker
ORD	Operations Requirements Documentation
ORD	Operational Requirements Document
OS	Operating System
OSCI	Open SystemC Initiative
OTA	Over The Air
PAR	Place And Route
PAT	Portability Assessment Team
PDR	Preliminary Design Review
PIM	Platform Independent Model
PLI	Programmable Logic Interface
PM	Project Manager
PMO	Program Management Office
POSIX	Portable Operating System Interface
PPVT	Post-Port Verification Test
PRF	Properties File
PRR	Porting Readiness Review
PSL	Property Specification Language
PSM	Platform Specific Model
QoS	Quality of Service
RF	Radio Frequency
RFP	Request for Proposal
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SAD	Software Assembly Descriptor
SCA	Software Communications Architecture
SCD	Software Component Descriptor
SDD	Software Design Description
SDP	Software Development Plan
SDR	Software Defined Radio
SEPG	Software Engineering Process Group
SoC	System on a Chip
SOW	Statement of Work
SPD	Software Package Descriptor
SPS	Software Product Specification

SRR	Software Requirements Review
SRS	Software Requirements Specification
STL	Standard Template Library
SVD	Software Version Description
SysML	Systems Modeling Language
TBM	Transactor-Based Modeling
TBV	Transactor-Based Verification
TDMA	Time Division Multiple Access
TRANSEC	TRANsmission SECurity
TRR	Test Readiness Review
UML	Unified Modeling Language
UUID	Universally Unique Identifier
VHDL	VHSIC Hardware Description Language
VHPI	VHDL API
VHSIC	Very High Speed Integrated Circuit
WAL	Waveform Acquisition Lead
WAT	Waveform Acquisition Team
WDS	Waveform Design Specification
WI	Waveform Integrator
WPA	Waveform Portability Assessment
WPAP	Waveform Portability Assessment Procedures
WPAR	Waveform Portability Assessment Report
WPG	Waveform Portability Guidelines
WPP	Waveform Porting Plan
WPR	Waveform Porting Report
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Appendix B Specific Terminology

This section defines terms that are specific to the context of waveform portability and its assessment.

Application Program Interface (API) – A specification that details the way in which two independent software components interact with one-another. An API describes how such a component may access a set of functions on another component without requiring access to the source code of the functions, or requiring a detailed understanding of the functions' internal workings.

Base Waveform – The base waveform is the original, non-portable waveform delivered by the waveform developer. This waveform is stored in the JTRS IR.

Coupling – Coupling or dependency is the degree to which any program module relies on another module. Loosely coupling interfaces in classes in C++, and functions in C minimizes the effects that changes in a module have on those modules associated with it.

Destination – See **Target**.

Development Environment – An environment used by the waveform developer to serve as a host for the waveform during the development process.

Environment – The collection of hardware, software, and peripheral devices that host an implementation of the waveform. Typical environments include multiple processors, specialized input/output (I/O) devices, an embedded cryptographic device, and SCA Operating Environment (RTOS, ORB, and CF).

Portability – The attribute of software that allows an application to run on two or more different environments.

Ported Waveform – The waveform software as it exists in a form that can execute on the target platform environment(s).

Porting – The act of moving a waveform by either rehosting or transporting from its original operating environment to a new and different one while retaining the original functional capabilities.

Rehost/rehosting – The act of modifying the waveform software to achieve the documented capabilities, requirements, and performance measures that drove the initial waveform design in the new platform environment. Rehosting is the most common form of porting.

Source – The textual representation of a computer program in a human readable form

Target (also Target Platform, Target Environment) – The environment to which the waveform software product is ported. Target platforms are fielded radios or functionally equivalent laboratory versions of those radios that provide the WI with greater visibility into the internal workings of the system.

Transport – Physically moving the waveform software and data to a new environment in which it will properly execute without change to either waveform or target platform source code.

Waveform Developer – An organization that creates the initial implementation of a waveform.

Waveform Integrator (WI) – An organization that ports a waveform to a specified platform. The WI may be the Government, the original waveform developer, or a third party.

Appendix C Acquisition Guidelines

Removed for public release.

Appendix D Modeling

Complex distributed systems such as JTRS waveforms are difficult to develop without some form of modeling involved. Many developers use modeling tools during the design process, but use of such tools for implementation has yet to become commonplace. Still, the possibility of improving portability using modeling tools to design and implement the waveform is significant. For this possibility to be realized fully, tools must exist that provide the designers and developers with the capability of designing a PIM that can be transformed by the tool into a PSM, and that can express the PIM in a manner that can be processed by other tools for other target platforms. Fortunately, a number of software vendors have recognized this need, and have created products and standards to fill this need. As more such tools are available, industries use of them will become more widespread.

In recent years, various organizations have proposed a number of standards related to software modeling. The Object Management Group (OMG), owner of the CORBA standard, has been particularly active in this area. The OMG has proposed and promulgated most of the standards discussed in this appendix; however, there are also other valuable standards, which often straddle the line between modeling techniques and software development methodologies.

Note: Waveform developers relying on modeling tools must be careful to avoid anything that would lock the PAT or subsequent WIs to the use of particular toolsets or techniques. This appendix is included to provide the reader with information on the current state of software modeling tools and techniques. This document does not endorse any particular tool or technique, nor does it require the use of such tools in the development of JTRS waveforms.

D.1 References

The references listed below are specific to this appendix.

- [D1] Cooley releases results of verification survey, http://www.edn.com/index.asp?layout=blog&blog_id=1480000148&blog_post_id=810008681
- [D2] Douglas C. Schmidt, Model-Driven Engineering, IEEE Computer, Vol. 39, No. 2, February 2006, pp. 41-47
- [D3] MDA Presentations and Papers, <http://www.omg.org/mda/presentations.htm>
- [D4] OMG Systems Modeling Language (SysML), OMG ptc/07-02-04, <http://www.omg.org/cgi-bin/doc?ptc/2007-02-04>
- [D5] Platform Independent Model (PIM) & Platform Specific Model (PSM) for Software Radio Components (also referred to as UML Profile for Software Radio) v1.0, OMG formal/2007-03-01, <http://www.omg.org/technology/documents/formal/swradio.htm>
- [D6] Open SystemC Language Reference Manual, IEEE 1666 SystemC Standard, <http://standards.ieee.org/getieee/1666/index.html>
- [D7] SystemVerilog, IEEE Standard 1800-2005
- [D8] Unified Modeling Language (UML), version 2.1.1, <http://www.omg.org/technology/documents/formal/uml.htm>
- [D9] UML Diagram Interchange Specification: OMG formal/06-04-04, <http://www.omg.org/cgi-bin/doc?formal/06-04-04>
- [D10] UML Profile for CORBA, v 1.0, OMG formal/02-04-01: http://www.omg.org/technology/documents/formal/profile_corba.htm

- [D11] UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms v1.0, OMG formal/06-05-02, http://www.omg.org/technology/documents/formal/QoS_FT.htm
- [D12] UML Profile for Schedulability, Performance and Time v1.1, OMG formal/2005-01-02, <http://www.omg.org/technology/documents/formal/schedulability.htm>
- [D13] UML Profile for System on a Chip (SoC), v 1.0.1, OMG formal/06-08-01, http://www.omg.org/technology/documents/formal/profile_soc.htm
- [D14] XML Metadata Interchange (XMI), v2.1, OMG formal/05-09-01, <http://www.omg.org/technology/documents/formal/xmi.htm>

D.2 Modeling Tool Exchange Formats

Achieving portability of waveform modeling requires that the model not only be portable between target platforms, but also that it is portable between modeling tools. This requires that such tools support a common data exchange format for model information. The OMG has proposed the XML Metadata Interchange (XMI) format for this purpose. XMI provides a mapping from Meta-Object Facility (MOF) to XML. XMI provides rules by which a schema can be generated for any valid XMI-transmissible MOF-based metamodel. In other words, XMI provides a standard way of interchanging models between MOF-based tools, such as Unified Modeling Language (UML). Waveform developers should be aware that recent versions of the XMI standard (2.0 and later) differ significantly from previous versions. To ensure compatibility with the largest number of modeling tool vendors, developers should only consider tools that support XMI 2.0 or later.

D.3 Modeling Languages

Unified Modeling Language (UML) has become the de facto standard modeling language in the software industry. UML defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object-oriented (OO) and component-based systems. A UML profile is a specification that defines a Domain Specific Language (DSL) that does one or more of the following:

- Identifies a subset of the UML metamodel
- Specifies “well-formedness rules” beyond those specified by the identified subset of the UML metamodel. “Well-formedness rule” is a term used in the normative UML metamodel specification to describe a set of constraints written in UML’s Object Constraint Language (OCL) that contribute to the definition of a metamodel element
- Specifies “standard elements” beyond those specified by the identified subset of the UML metamodel. “Standard element” is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value, or constraint
- Specifies elements, semantics, syntax, and constraints expressed in natural language, beyond those specified by the identified subset of the UML metamodel
- Specifies common model elements, expressed in terms of the profile

Applicable UML Profiles for JTRS are:

- The PIM & PSM for Software Radio Components (also referred to as the UML Profile for Software Radio)
- The OMG Systems Modeling Language (SysML), which defines a new general-purpose modeling language for systems engineering
- The UML Profile for CORBA, which defines a standard means for expressing the semantics of CORBA IDL using UML notation

- The UML Profile for Modeling QoS and Fault-Tolerance Characteristics and Mechanisms, which defines a set of UML extensions to represent Quality of Service and Fault-Tolerance concepts
- The UML Profile for Schedulability, Performance, and Time, which defines standard paradigms of use for modeling of time-, schedulability-, and performance-related aspects of real-time systems in a standard way
- The UML Profile for System on a Chip (SoC), which defines a standard way to model system level design, software algorithms, and hardware architecture, and can be transformed into SystemC, SystemVerilog, and VHDL

D.4 Model Driven Development

Model Driven Development (MDD) and Architecture (MDA) are technologies proposed by the OMG as the foundation of their software modeling strategy. The References section above lists papers describing MDD and MDA. At the core of MDD are models that are used for the understanding, definition, design, construction, implementation, and deployment of systems. The models can adopt different viewpoints, such as platform and application perspectives, along with the degree to which they involve technology information.

Model development has been performed in the past where models have been created at different system development phases, such as requirements, design, and implementation. MDD differs from traditional modeling in that models progress from design to PIM implementation, and then on to PSM implementation through automatic transformation within the tool(s). In particular, design PIMs are living artifacts that are not created once, and then forgotten about, in preference to code written in third-generation languages. Thus, the designs and implementations remain synchronized. The more recently developed modeling tools have the capability of automatically including the required infrastructure from applicable standards within the model.

Some possible transformations for a waveform PIM into PSMs include:

- Source code generation
 - Different source code generators such as C, C++, VHDL for waveform's component implementations
- Complete executable test code generation
- SCA XML files based upon SCA DTDs
- Makefile compilation system for host or target RTOS.

The transformation capability of waveform PIM into PSMs will vary by tool vendor, such as:

- The types of PSMs generated, such as test code, language source code, SCA XML files, and Makefile compilation system
- The quality of the PSMs
 - The level of SCA functionality of the source code that is generated
 - Quality and documentation of the generated code

While the waveform is undergoing transformation into a PSM, it should also be optimized for maximum performance and efficiency on the target platform. Such optimization should allow for manual control of parameters, such as memory layout, etc., in order to ensure repeatability. This optimization is likely to reduce the portability of the *ported* waveform, but this is acceptable, as the *ported* waveform does not have to undergo subsequent ports.

D.4.1 Modeling Types

Modeling for a waveform involves many different facets depending on the type of component under development. Until recently, most modeling tools tended to focus on only one type of CE, requiring the developer to use different tools for GPP, DSP, and FPGA development. This made it impossible to model the entire system at one time. As newer tools have come onto the marketplace, that situation has improved, allowing a developer to model the entire system at once, and move it through the various transformations as a cohesive design, thus supporting an MDD approach. However, as stated earlier, the use of these newer tools is not yet widespread.

D.4.1.1 Platform Independent Modeling

As stated in Section D.4, a PIM is a model that describes the definition of some element independent of a specific technology. An example of PIM would be a waveform application model that captures the application's components along with the inter-component connections (QoS), and component definitions. A component definition could consist of properties, supporting interfaces, subscriber and provider ports, state behavior, and algorithmic behavior. A waveform PIM could also provide various use-case scenarios depicting the design interaction of the application's components along with any timing constraints. There may be multiple PIMs created to capture the logical design of an element, each giving a different viewpoint or perspective.

A properly detailed and documented PIM of the entire waveform may well be the most useful way to describe its design as it concentrates a great deal of information into a form that can be understood by engineers and manipulated by modeling tools. An engineer can use a properly developed PIM to model adaptations to new platforms, as well as insertions of new features or technologies (through platform-specific optimizations).

The primary benefit of a PIM is that an element definition is created once and remains consistent. Likewise, a PIM can be applied to new/different technologies by transforming the element's parts to the new/different technology while still adhering to the element's definition. In the JTRS program, for instance, a waveform PIM can be used by future radio procurements that require that waveform's characteristics, thereby reducing cost and time to market. The intellectual property and value-added capabilities provided by a contractor's implementation for a particular radio constrain the transformation of the waveform PIM into their waveform PSM for the radio's physical communication channels.

As mentioned in Section D.3, SysML and UML Profiles, such as SoC, Software Radio, and QoS, would be good candidates for developing platform and waveform PIMs. Other modeling for signal processing may involve proprietary PIM formats, such as MatLab.

Modeling tools typically utilize proprietary technology for their implementation. For example, the representation of UML internally to an MDD tool is a vendor's proprietary technology. To avoid vendor lock-in, therefore, it is essential to be able to exchange models between toolsets using a standard format, such as XML.

D.4.1.2 Platform Specific Modeling

As stated in the introduction to D.4, a PSM is a model that describes the definition of some element dependent on a specific technology, such as C++, XML, or CORBA. Examples of a SCA PSM would be a waveform C++ detailed design using UML or a component's XML descriptors. Within the MDD context, the tool should automatically transform a PIM into a PSM. This transformation is extremely powerful as a development aid. Tool-automated transformations allow for quick insertion of platform-specific optimizations, new features, and new technologies in an environment that also supports testing of the modifications to the model.

Most UML tools support modeling for specific third-generation languages, such as C/C++ and Java, but the UML language profile being used is not standardized. In the OMG, there are no standard UML profiles for third-generation languages. A UML C++ model produced from tool vendor 'A' may therefore be usable by another UML tool vendor 'B', provided the UML C++ profile is not restrictive from license usage and conforms to XMI, as stated in D.1. This situation will allow developers to review the C++ model using a different UML tool, but be unable to generate code from the model since the UML tool may not have C++ generators for this type of UML profile. The adoption of standard UML profiles would resolve this problem and promote model reuse across the JTRS program.

D.4.1.3 System / Algorithmic Level Modeling

Three standard IEEE HDLs are Verilog, VHDL, and SystemC. Verilog and VHDL are the oldest and have been around since the 1980s. According to a survey article from *EDN* magazine, Verilog is more widely used than VHDL. SystemC is the newest and least popular, although its use has been rising in the industry. The *EDN* article also stated VHDL is used predominantly by US military contractors and some European companies.

IEEE 1800 SystemVerilog (<http://www.systemverilog.org/>) is the industry's first unified hardware description and verification language (HDVL) standard. SystemVerilog is a major extension of the established IEEE 1364 Verilog language that is based upon C and Ada.

IEEE 1076 VHDL is based upon the Ada Language. There is currently an updated VHDL standard draft approved by IEEE that was to have been completed in 2007, but is still under development. This draft amends the 1076 2002 with VHDL API, known as VHPI, the other child standards (1164, 1076.2, 1076.3) that addressed issues in 1076, and a subset of the Property Specification Language (PSL).

IEEE 1666 SystemC 2.1 is based upon the C++ language, which was initially developed by the Open SystemC. The Open SystemC Initiative (OSCI) (<http://www.systemc.org/>) is an independent not-for-profit organization composed of a broad range of companies, universities and individuals dedicated to supporting and advancing SystemC as an open source standard for system-level design.

All HDLs have similar capabilities for describing hardware structure (e.g., module, channel, protocol) and behavior (e.g., process, algorithm). They also have differences in the capabilities being offered for System, Algorithmic, Register Transfer Level (RTL), Logic, and Gate. Modern HDLs for high-level design address both hardware and software together.

Since there are multiple standard HDLs available, it is very important to be able capture the system/subsystem or hardware design independent of a specific HDL. One should consider capturing the system design based upon the OMG UML SoC. The UML SoC can be used to model hardware system level design as a PIM independently from a HDL. The PIM then can be transformed into a HDL such as SystemC or SystemVerilog. The OMG UML SoC specification did not mention VHDL, although since it has similar concepts as SystemC and SystemVerilog, it should be possible to transform a UML SoC PIM into VHDL.

In the DSP domain, modeling tools, such as MatLab, provide both a highly productive system-level verification environment and an efficient path to implementation for standard DSP processors. Built-in abstractions liberate the designer from the strict modeling style guides that are required by general-purpose languages, allowing the representation of large design objects with a high degree of efficiency.

D.4.1.4 Constraint Modeling

In addition to describing the relationships between model elements, it is often useful to apply constraints to a model, for example, to specify properties, such as QoS. There are various approaches to applying constraints to MDD applications, such as Object Constraint Language (OCL), Third-Generation Languages, and Prolog.

OCL is a declarative language whose type constraints provide an alternative way of specifying constraints that is often more practical and more conducive to better programming, e.g., simplifying iteration through lists. In general, it is easy to write simple constraints using OCL. As soon as constraints get more complex, however, OCL becomes hard to read and maintain.

Another approach is to write the constraints in a third-generation language, such as Java, C++, or C#. For simple constraints, developers end up writing a little bit more code than OCL. For complex constraints, however, third-generation language code tends to be more readable than using OCL. Third-generation languages also have good tool support. Third-generation languages are also good when developers need to check non-traditional constraints that involve several model elements, need temporary variables, etc., which are hard to do with the OCL model.

Yet another approach is to use a rule-based language, such as Prolog, and provide the constraints in a model-independent way. This approach combines the benefits of the other two approaches. It is easier to write some types of complex rules in Prolog (assuming Prolog expertise, of course). The main problem with Prolog is that it is hard to integrate with conventional programming languages and tools.

D.4.2 Source Code Generation

Source code generation involves synthesizing language-specific artifacts for a model element. This has typically been done for CORBA IDL or C++ source code development, but some newer tools also include test code and makefiles as well. Source code generation typically involves multiple modeling tools for SCA component infrastructure, a component's detailed design, and algorithms. Generated test code should remain with the model and the generated source, and should be automatically updated along with them.

D.4.2.1 Coding Guidelines

Source code produced from modeling tools should support the coding guidelines stated in the body of this document. Code generator tools rarely provide documented code, but it would be a plus for code generators to supply supplemental documentation (comments) to make it easier to understand the generated code. For example, a tool may generate comments for the validation logic of a component's properties or for a component's names involved in an assembly's connection definition. Such a capability is mandatory in the case of secure code generation. This could be in the form of a pass-through of comments from model to source code.

D.4.2.2 Portability Guidelines

Source code produced from modeling tools should comply with the portability guidelines stated in the body of this document. In particular, no vendor-specific headers should be used and the code should be RTOS- and ORB-neutral (e.g., no vendor-specific macros or API calls). The component's business logic implementation should be a separate file from the SCA component CORBA servant code. This design allows a component's implementation to evolve independently from the component infrastructure.

D.4.2.3 Language Support

SCA waveform and platform components are typically written in C++ language for GPPs, C language for DSPs, and VHDL for FPGAs. The more of these languages that a given tool supports, the more useful it is likely to be for waveform modeling and development.

D.4.2.4 Design Practices/Patterns

The JTRS program involves many waveforms developed by different companies. These waveform designs are rarely done the same way by different companies or even within the same company. MDD provides the opportunity to develop a repository of model elements that are reusable across waveform and platform components by applying the same design patterns. Such standardized components, if previously certified

by the appropriate evaluating entity, could also expedite subsequent recertifications when implemented as parts of new applications. Design patterns have been used for some time at the OO programming level. When utilizing MDD, however, design pattern reuse is at the component level, where new waveforms can be rapidly built using existing design patterns [e.g., encoders, decoders, modulation, filters, Carrier Sense Multiple Access (CSMA), Time Division Multiple Access (TDMA)].

The standardization of JTRS APIs and devices/services is a way of establishing design patterns at the service component level, which is where maximum portability will occur. The design patterns within waveforms are probably the next most beneficial level to investigate for possible standardization.

Another benefit of MDD is achieving a high level of consistency between component implementations. This consistency is more easily achieved because the same transformation design pattern generators may be used for each component generation. Enforcement is not left to developers, but is handled automatically by MDD tools. The benefit of automated tool enforcement is that SCA validation is performed once, since the same transformation is used for all components, so validation time is spent more appropriately on the waveform.

D.4.2.5 Platform Specific Build Generation

Modeling tools vary in their support for creating makefile projects that can compile and link the generated source code for different processing environments: RTOS, ORB, processor, and language. As mentioned earlier, a waveform does not have the same implementation for all its components. It is, therefore, important to have the capability to define different processing environments and associate these processing environments to components. The automatic creation of makefile projects that work with these processing environments, RTOS compilers, linkers, libraries, and ORB libraries and IDL compilers is beneficial in that it relieves the developer of these tasks; however, the availability of such a capability is limited to only a few currently available toolsets.

D.4.2.6 Document Generation

Design documentation can be viewed at two levels: the PIM and PSM. At the PIM level, the documentation is analogous to the traditional preliminary design. The PIM captures the logical design for a component or application, and all PSMs adhere to the PIM. The design information for a component PIM can contain statechart behavior, algorithm behavior, relationships of the component's ports along with ports' QoS and interfaces, and use-case design scenarios.

The design information for an application PIM can contain the assembly of the components along with use-case design scenarios with timing information. The application PIM should be the complete application definition, not just for certain types of components whose implementations are on a GPP.

The PSM level is analogous to the detailed design. A component PIM may have different PSMs, such as C++, C, or VHDL. For each type of component PSM, the design pattern can capture once how the PIM is transformed into a specific PSM. The design information for a component PSM can contain provider port implementation and use case detailed design scenarios.

D.4.3 Test Management

As with Requirements Management for architectural requirements, test cases can be pre-built into a project database. Along with test code for these test cases, a component's definition will determine the number of architectural test cases and test code for a given component design. So, for example, if a component has two SCA *uses* ports (CF Port interface), the same set of *uses* port test cases are applied against the component's *uses* ports. The *uses* test code may vary slightly in the area of connect or disconnect since the types of connections are probably different.

D.4.4 Requirements Management

Requirements Management is usually a manual effort of associating requirements to design and test cases, but some of these activities can be automated depending on the type of modeling tool being used. For architectural requirements like SCA [4], these requirements can be automatically associated with component designs. As components are designed at the PIM level, and based upon the type of component they are, the associated requirements can be automatically set in the requirements database. As the component's PIM designs are transformed into implementation design, the requirements flow down to the implementation design at an operational level. Likewise, the requirements associated with a component's interfaces can automatically flow down to, and be associated with, the interfaces' implementation design.

As an example, the requirements associated with an SCA *CF::Resource* interface can be automatically associated with a waveform *Resource* PIM component. As the waveform resource PIM component is transformed into C++ source code, the SCA *CF::Resource* interface requirements are associated with the resource interfaces in the C++ source code. Each waveform Resource PIM component and its implementation source code trace to the same SCA requirements database module, all of which can be accomplished automatically without operator involvement within tools that conform to MDD.

D.4.5 Configuration Management

Configuration Management for MDD involves the configuration control of the model elements, but not necessarily the artifacts generated from a model that have no need for operator intervention (such as component's subscriber--uses--port source code or entry point source code). The IDL compiler and compiler switches that were used to generate the code are important to know as stated in Section D.4.7 below. At a minimum, the MDD artifacts that must be controlled are the CORBA IDL files (model or actual IDL), application model files, and an application's implementation files containing the business logic.

D.4.6 Simulation and Synthesis

Simulation of a design or implementation can occur from a model, from HDL, or from actual waveform language code on a host SCA OE. In addition, simulation of a design can occur at multiple, different levels such as system, subsystem, and component levels for hardware and/or software. Synthesis of a design is the actual conversion of a high level physical description of a design into the primitive blocks and connections found in silicon devices, such as FPGAs and Application-Specific Integrated Circuit devices (ASICs).

At this time, simulation at the model level is vendor proprietary, which means these types of models may not be importable into another vendor's simulation tool, or, if imported into another vendor's simulation tool, may not simulate. UML Executable, a UML profile, is currently in the submission process at the OMG; however, UML Executable is over a year away from formal adoption. Until finalization and adoption of the UML Executable standard, or a decision by the JPEO to standardize on a particular set of tools, model-level simulation will not be portable between JTRS waveform development environments.

In the past, HDL simulation was mostly performed from a hardware perspective but with modern HDLs (e.g., VHDL) simulation can now involve software. This is called co-verification of hardware and software and can decrease the development time of a project by a significant amount. Co-verification usually involves software running on a processor that communicates with functions embedded in the hardware of an FPGA or ASIC. A typical co-verification methodology is to write a software model of the hardware functions in C or C++ and then run these with the software in order to do system level verification and analysis. The bus level activity of the software models is then captured and converted into test vectors that could be run in the hardware simulation environment. The functional description of the hardware is passed to the hardware design group who then produce a synthesizable RTL version of the design. The test vectors

are used to verify that the RTL behaved the same as the software models used in the system analysis. This methodology makes it difficult to quickly simulate changes to the software and the runtime of the software is limited by the maximum size the static test vectors. Attempts were made to improve on this method by creating Bus Functional Models (BFM) for the processor that could be run under hardware simulation. This moves the simulation to a higher level of abstraction making it easier to simulate software changes; thereby, decreasing the size of the stimulus needed for the simulation. The problem here is that the BFM is only a model and may not behave in the exact way that the actual processor does. In addition, it is difficult to convert the software functions into stimulus for the BFM. These factors increase design risk and development time.

The introduction of Transactor-Based Modeling (TBM) has increased the efficiency of verifying hardware and software together. A transactor is able to synchronize the data and commands passing between the software running on a processor and the hardware running under simulation. The transactor is instantiated in the hardware simulation environment and communicates with the hardware under test via an RTL interface that models the processors physical I/O and bus protocol. The transactor communicates with the external software using software protocols passed through the simulator's Programmable Logic Interface (PLI). Every simulator vendor provides their own PLI but they all use a set of standard software calls. The transactor has the ability to pause the simulation under program control of the software, thus keeping the hardware and software in lockstep throughout the simulation run. This layer also takes care of inter-process communications via shared memory between the software test process and the simulator. A shell program coordinates starting the simulator and software test, and then takes care of test results, message logging, and orderly simulation shutdown when errors occur. Transactor-Based Verification (TBV) is a powerful method to achieve co-verification, but requires time and specialized knowledge to create the transactors and to work with the PLI interface. Several companies provide tools and environments for TBV that ease the development effort. These include the VERA testbench automation product from Synopsys and the QuickBench verification tool from Forte Design Systems. Transactor-based processing is currently the most popular method used for co-verification.

Another method for co-verification (used mainly in ASIC work) is emulation. The emulator is a piece of hardware that contains a processor, large amounts of memory and several large FPGAs tied together through a mesh communication network. The RTL hardware design is mapped into the FPGAs and the mesh network is used to tie the memory and FPGAs together, as if the design were on one large chip. This enables the design to communicate with test software running on the embedded processor or with an external processor via a standard interface. The emulated design runs at a fraction of the final ASIC speed but it is still many times faster than running on a simulator. This allows long software run times that detect low level as well as system level issues. The drawbacks to emulation are the cost of the emulator hardware and difficulty of mapping a complex ASIC design into several FPGAs.

There are many different simulators for simulating VHDL, SystemC, and SystemVerilog in the industry. There should be consistent usage of one HDL in order to promote reuse and reduce costs across projects. For the JTRS program, VHDL has so far been the HDL of choice, and it is often useful to simulate the executable software on a host environment before execution on a target platform. Instruction Set Simulators (ISS) and RTOS simulators are available that allow software to be verified on a workstation using source level debuggers. This provides the benefit of verifying the functionality of some of the waveform components along with their interactions.

Languages such as SystemC and System Verilog allow engineers to design and verify at the system level and create physical RTL code automatically for the hardware parts of the system. This is desirable because it removes the step where the hardware engineer creates RTL from a specification of functions that have already been verified in behavioral software at the system level. There are several "behavioral synthesis" tools in existence, with some of the best-known being from Forte Design Systems and Celoxica. These tools support SystemC and C/C++ and generate RTL from the behavioral code. The user can then follow the standard RTL to synthesis flow. A main concern with such tools is the quality of the machine-generated

RTL. Formal verification must be performed between the behavioral code and the RTL to ensure equivalence. Without formal verification, there is no way of knowing that the design tested at the behavioral level is the same as the design synthesized at the RTL level. Some verification tool vendors claim to be able to verify designs where the synchronous elements are changed, e.g., pipelining or clocked delay stages added. These advances may close the formal loop, but credible statistics are needed on the quality of the RTL generated.

There is much work being done on formal verification and behavioral-to-RTL or direct behavioral-to-gates compilation. The compiler-generated RTL is improving and the formal verification tools are becoming more trustworthy. At this time, however, tools using C/C++/SystemC and System Verilog are best used for architectural simulation, while human generated RTL is best used for physical design and functional verification.

D.4.7 Development Environment

D.4.7.1 Host Development Environment Support

Before developers actually test a waveform, or deploy it onto a target platform, they should first build the components on a host environment and verify application execution in that environment. For one user to build an application that another engineer has developed, one needs (at a minimum) the following list of artifacts:

- The makefile project that was used to build the application code
- Environment variables that must be set up and used within the makefile project
- Identification of the target in the makefile project is used to build the application code that invokes the other makes
- Source code to be compiled, linked, and tested. It is advisable to keep the source code separate from the makefile project so other processing environments (VxWorks, Linux, etc.) can easily be built.

Processing environment description documents the host operating system compiler switches used, along with libraries, paths, linker switches, and so forth. An example of the type of information needed for a Processing Environment Build Description is as follows:

- Source language (e.g., C, C++)
- Processor (x86, PowerPC, etc.)
- RTOS build environment
 - RTOS name (e.g., VxWorks, LynxOS, Linux) along with version
 - Archiver command name used to create static libraries along with flags
 - BASE PATH added to the environment PATH variable
 - Compiler command, flags, libs, include paths – this is the compiler and/or linker command name for compiling source and link object images
 - Linker command name, flags, path, and libraries
 - Object and/or shared extension suffix name for the object files (e.g., “.o”, “.so”)
- ORB environment
 - ORB name is the name of the CORBA ORB along with version
 - CORBA include and libs path – this is the file path for the CORBA COTS header files
 - CORBA link switches are switches for linking the CORBA object files
 - IDL compiler command is the IDL compiler command name along with flags

D.4.7.2 Target Development Environment

As described for a host environment, the same information is needed for the target environment, and the same source code used for host development should be compilable across RTOSs and ORBs where compatible. Deliverable items could be the makefile project that is used to build this environment. Additional information would be:

- The RTOS features that were used for the target platform. Most RTOS are configurable by an end-user to the set of features or capabilities (e.g., web browser, communication protocols) that form the RTOS for the specific platform.
- The ORB features that were used for the target platform. Most ORBs are configurable by an end-user to the set of features or capabilities (e.g., types supported, usage of the any keyword, Real-Time Support, compact/micro profiles) that form up the ORB for the specific platform.
- The board along with its Board Support Package.

D.4.7.3 Third Party Tools Integration

Common tooling frameworks help ease the task of integrating disparate development and simulation tools, one such framework being Eclipse (<http://www.eclipse.org>). Such frameworks allow several disparate tools to be presented as a single, integrated toolset with a common user interface. Standardizing on a tools framework for SDR development is beneficial as it allows the use of an IDE that spans the full SDR development lifecycle. Waveform developers should investigate the tool sets available for the optimum degree of integration.

In lieu of integrations via a common tooling framework, one may choose more loosely coupled integrations. These integrations, for example, are usually implemented via the adoption of a particular set of design processes. These design processes define the tooling workflow and the handoff of artifacts from one tool to another. These types of integrations will not have the same smooth flow of information from one tool to the other as offered by the common, integrated frameworks.

It should also be noted that additional integration between modeling tools could be achieved when meta-model definitions, a DSL, are shared between modeling tools. By using XMI as an exchange format, the modeling tools can actually then exchange modeling files between each other. This allows particular modeling tools to focus on aspects of the waveform design that the tools are designed to solve. For example, UML modeling tools could exchange data with SCA design tools directly. Each tool would open these files using editors specifically designed either to support either OO Analysis and Design, or SCA component-based design. The two tools would manipulate the same object model allowing it to be consistently maintained regardless of the tool used. When available, this style of integration is available across modeling tools and should be taken advantage of as it reduces duplication and increases the accuracy of modeling information during system design.

D.4.7.4 Radio Platform Integration

When choosing modeling tools for SCA development or when creating an SCA development IDE by the integration of development tools, a number of radio platform integration issues should be also considered. Among these are the download of XML and target component executables, target and model-level debugging, OE monitoring, SCA Application monitoring, and deployment visualization.

Downloads of XML and target executables must use SCA compliant interfaces including the CF::File, CF::FileSystem and CF::FileManager interfaces of the SCA to move file onto the target radio hardware.

Support for debugging on the target radio platform is also very important during integration. At a minimum, the debugging facilities of RTOS vendors should be incorporated into a development IDE. If

possible, the incorporation of model-level debugging facilities will further raise the level of abstraction and aid developers in locating bugs, race conditions and throughput bottlenecks. Several modern modeling tools allow users to set breakpoints and watchpoints at the model level and then generate the instrumented code required to support this type of debugging. Given the complexity of SCA-based systems and the amount of automatic code generation produced by IDL compilers, UML tools, and SCA development tools, this type of debugging is becoming more and more important to SCA software developers.

SCA software developers also need to be able to monitor the OE in order to deploy, tear down, and observe the state of the SCA framework running across all the physical hardware in the radio, and such capabilities should be part of the IDE. Along with these facilities, software developers of SCA-based systems also need applications that monitor the deployment and execution of instantiated waveforms. Graphical versions of these tools, which visually provide deployment and configuration information, should be preferred over command-line text based systems as textual representations tend to be complex and cumbersome to use.